PHYSICS DEPARTMENT

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Quantum Science & Technology

Machine Learning Potentials with Long Range Interaction

Aaron Schulz

PHYSICS DEPARTMENT

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Quantum Science & Technology

Machine Learning Potentials with Long Range Interaction

Potentiale des Machinellen Lernens mit Fernwirkung

Author:	Aaron Schulz
Supervisor:	Christian Mendl
Submission Date:	05.10.2024

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Dr. Christian Mendl, for the continuous support of my master's thesis and his motivation, support, advice, and knowledge.

His guidance helped me all the time during the study and writing this thesis.

Abstract

This thesis investigates the development and application of machine learning (ML) potentials that incorporate long-range interactions to enhance molecular simulations. Traditional ML models, such as neural networks, have been widely used for short-range interactions, effectively modeling local atomic environments. However, they often fail to capture critical long-range interactions that are essential in accurately describing the behavior of large molecular systems. In this work, a new extension of an existing ML approach is presented, leveraging the Fast Multipole Method (FMM) to efficiently approximate long-range interactions within neural network frameworks. This reduces the computational complexity typically associated with direct pairwise calculations.

The proposed model integrates a grid structure, which allows distant atoms to be represented by their collective centers of mass, significantly improving scalability while maintaining accuracy. The method is benchmarked against conventional neural networks and demonstrates improved performance in predicting molecular energies. Results show that this method yields a slightly lower Mean Absolute Error (MAE) compared to models that ignore such interactions. Additionally, suggestions for future improvements are made to further enhance model performance.

Kurzfassung

In dieser Arbeit wird die Entwicklung und Anwendung von ML-Potenzialen (Maschinelles Lernen) untersucht, die Wechselwirkungen über große Entfernungen berücksichtigen, mit dem Ziel, molekulare Simulationen zu verbessern. Herkömmliche ML-Modelle, wie neuronale Netze, wurden häufig für Wechselwirkungen über kurze Entfernungen verwendet und modellieren effektiv lokale atomare Umgebungen. Sie berücksichtigen jedoch oft Wechselwirkungen über große Entfernungen nicht, die für die genaue Beschreibung des Verhaltens großer molekularer Systeme unerlässlich sind. In dieser Arbeit wird eine neue Erweiterung eines bestehenden ML-Ansatzes vorgestellt, bei der die Fast Multipole Method (FMM) zur effizienten Annäherung von Interaktionen mit großer Reichweite im Rahmen neuronaler Netze eingesetzt wird. Dadurch wird der Rechenaufwand reduziert, der normalerweise mit direkten paarweisen Berechnungen verbunden ist.

Das vorgeschlagene Modell integriert eine Gitterstruktur, die es ermöglicht, weit entfernte Atome durch ihre kollektiven Massenschwerpunkte darzustellen, was die Skalierbarkeit bei gleichbleibender Genauigkeit erheblich verbessert. Die Methode wird mit herkömmlichen neuronalen Netzen verglichen und zeigt eine verbesserte Leistung bei der Vorhersage molekularer Energien. Die Ergebnisse zeigen, dass diese Methode im Vergleich zu Modellen, die solche Wechselwirkungen ignorieren, einen geringfügig niedrigeren mittleren absoluten Fehler (MAE) aufweist. Darüber hinaus werden Vorschläge für zukünftige Verbesserungen gemacht, um die Leistung des Modells weiter zu steigern.

Contents

1. Machine Learning

Machine learning (ML) has emerged as a powerful tool for solving complex computational problems across various fields, from image recognition to molecular simulations. At its core, ML involves training models to identify patterns and make predictions based on data. One of the most successful ML techniques is neural networks, which are inspired by the structure of the human brain and capable of modeling highly non-linear relationships between inputs and outputs.

Neural networks consist of interconnected layers of neurons, each neuron functioning as a mathematical unit that processes input data. A single neuron mimics the behavior of biological neurons, receiving weighted inputs, applying a bias, and producing an output through an activation function. This setup allows the network to approximate complex functions, and when multiple neurons are stacked across layers, they can model intricate data patterns. It has been proven that neural networks with at least one hidden layer are universal approximators, meaning they can approximate any continuous function to arbitrary accuracy, given sufficient resources [1].

However, training these networks effectively requires several important components. Non-linear activation functions, such as the Rectified Linear Unit (ReLU) or sigmoid, ensure that networks can learn beyond linear relationships. Optimization methods like gradient descent, including its stochastic variant, are used to iteratively adjust the weights of neurons to minimize a loss function, which measures the difference between the predicted and actual outcomes. More sophisticated techniques like the Adam optimizer adaptively adjust learning rates to improve training stability and convergence [2][3].

Despite the power of neural networks, challenges like overfitting and vanishing gradients must be carefully managed. Techniques such as dropout and layer normalization help regularize the model and ensure effective training by preventing neurons from co-adapting too much or by stabilizing input distributions across layers [4][5].

In this chapter, we will explore the foundational concepts of machine learning, with a specific focus on neural networks, and introduce key methods and techniques that enable the successful training of these models.

1.1. Single Neuron

The structure of a single neuron of a neural network is inspired by the structure of the human brain. A neuron in the human brain consists of dendrites, which act as signal inputs, and axons, which act as signal outputs. Neurons are connected via dendrites and axons (Figure 1.1). In the mathematical model of the neuron, the inputs (dendrites) x_i are multiplied by the

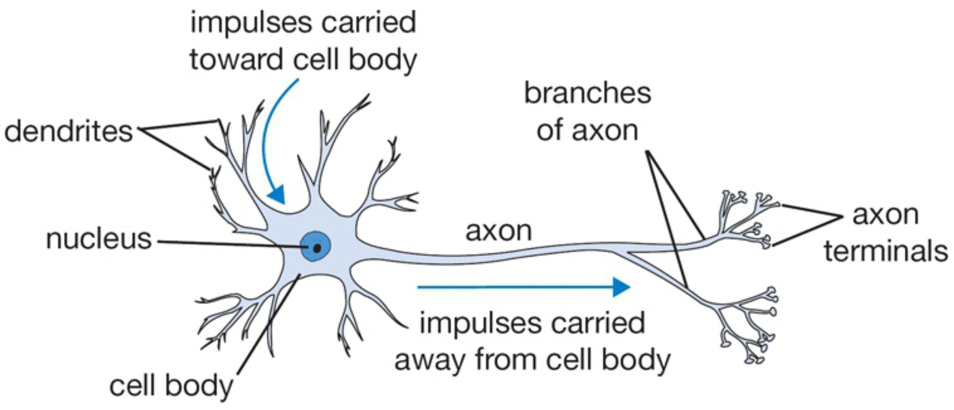


Figure 1.1.: Biological Neuron

weights of the neuron w_i and then added together. Here, w_i is the weight of the i-th input. If the value of this sum is greater than a threshold value θ, the neuron fires.

$$y = \left\{ \begin{array}{ll} 1, & \sum\limits_i w_i x_i + b \geq \theta \\ 0, & \text{otherwise} \end{array} \right\} \tag{1.1}$$

y is the output of the neuron, and b is called the bias, which is also a parameter. Without the bias, the output of a neuron would always be 0 (regardless of the w_i) if all $x_i = 0$. In practice, a threshold is not used due to considerations of continuity and differentiability; instead, the sum is subsequently used as the input to a so-called activation function f (see Section 1.2):

$$y = f\left(\sum_i w_i x_i + b \right) \tag{1.2}$$

It can be seen that a single neuron implements a linear classifier [1].

1.2. Activation Functions

To ensure that stacking multiple neurons does not result in a purely linear function, activation functions or non-linearities are introduced. These are scalar functions with a single number as input [1]. In the following sections, several activation functions will be presented, along with a discussion of their advantages and disadvantages.

1.2.1. Sigmoid

The sigmoid function is mathematically expressed as

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{1.3}$$

The real-valued input x is mapped by σ to the interval [0,1]. Large negative numbers are mapped to 0, and large positive numbers are mapped to 1. Historically, the sigmoid function was frequently used because it closely resembles a threshold function. However, it has two major disadvantages:

1. When the neuron's output is close to either 0 or 1 (i.e., it saturates), the gradient in these regions becomes nearly zero. During backpropagation, this small gradient is multiplied by the gradients of subsequent layers, leading to a phenomenon known as the "vanishing gradient problem." This issue can severely hinder the learning process, as almost no signal is passed back through the neuron to adjust the weights effectively. Moreover, careful weight initialization is crucial for sigmoid neurons to avoid saturation from the start, as large initial weights can cause most neurons to saturate, leading to minimal learning.

2. When neurons receive non-zero-centered data in later layers of a neural network, it can affect the dynamics of gradient descent. Specifically, if the input data to a neuron is always positive, the gradient on the weights during backpropagation will consistently be either all positive or all negative, potentially causing zig-zagging in the gradient updates. This can slow down the learning process. However, when gradients are averaged across a batch of data, this issue is somewhat mitigated, making it less severe than the vanishing gradient problem associated with saturated activation functions.

[1]

1.2.2. ReLU

The Rectified Linear Unit (ReLU) function is defined as $f(x) = \max(x, 0)$. The ReLU function has the advantages of accelerating convergence during training with Stochastic Gradient Descent and being a computationally efficient operation compared to the sigmoid function. However, a neuron using the ReLU function can irreversibly "die" (known as the "dying ReLU" problem), meaning it may no longer activate for any data point. This can be avoided by using the appropriate learning rate. [1]

1.2.3. ELU

The Exponential Linear Unit (ELU) function is defined as

$$f(x) = \begin{Bmatrix} x, \quad x > 0 \\ \alpha(\exp(x) - 1), \quad x \leq 0 \end{Bmatrix} \tag{1.4}$$

where α is a hyperparameter that controls the saturation point for negative inputs. One key advantage of ELUs over ReLUs is their ability to produce negative values, which helps to push the mean activations of neurons closer to zero, thereby reducing the bias shift effect. This leads to faster learning by bringing the gradient closer to the natural gradient. Additionally, for large negative inputs, the ELU function saturates, which can make the model more robust to noise and provide a better generalization. In contrast, ReLUs can suffer from the ”dying ReLU” problem, where neurons get stuck during training and output zero for any input. However, ELUs have the disadvantage of being computationally more expensive than ReLUs due to the exponential calculation, which can slightly increase training time in practice. [6]

1.3. Neural Networks

Neuronal networks can be represented as computational graphs (see Figure 1.2). The neurons are the nodes, which are connected by edges representing the flow of computations. The output of one neuron can serve as the input to other neurons. If the graph is acyclic, these networks are called feedforward neural networks because information flows in a single direction, from input x, through various layers to output y, without any feedback loops. When feedback is incorporated, the network becomes a recurrent neural network [2]. This thesis focuses exclusively on feedforward neural networks.

The primary goal of a neural network is to approximate a target function f^*, such as mapping an input x to an output y. In this process, the network learns the parameters θ to optimize the function approximation. x and y are often vectors, meaning they consist of multiple entries.

The term ”network” is used because these models are composed of multiple interconnected layers, each representing a function. For instance, a network might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$, forming a chain where $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. Each function $f^{(i)}$ corresponds to a layer, and the depth of the network refers to the total number of layers. The final layer is the output layer, and the intermediate ones are the hidden layers, which the learning algorithm must optimize.

Each hidden layer is typically vector-valued, meaning it consists of several neurons that operate in parallel. Each unit processes inputs from the previous layer, computes its activation, and passes this result to the next layer. [2]

One type of layer is the so-called ”fully-connected layer” or ”dense layer”. In these layers, all neurons from two consecutive layers are pairwise connected. Neurons within the same layer, however, are not connected (see Figure 1.2).

It has been shown that neural networks with at least one hidden layer are universal approximators. This means that, given any continuous function $f(x)$ and any small error margin $\epsilon > 0$, there exists a neural network $g(x)$ with one hidden layer (using a non-linearity like a sigmoid) such that for all x, the difference between $f(x)$ and $g(x)$ is smaller than ϵ. In other words, a neural network can approximate any continuous function to an arbitrary degree of

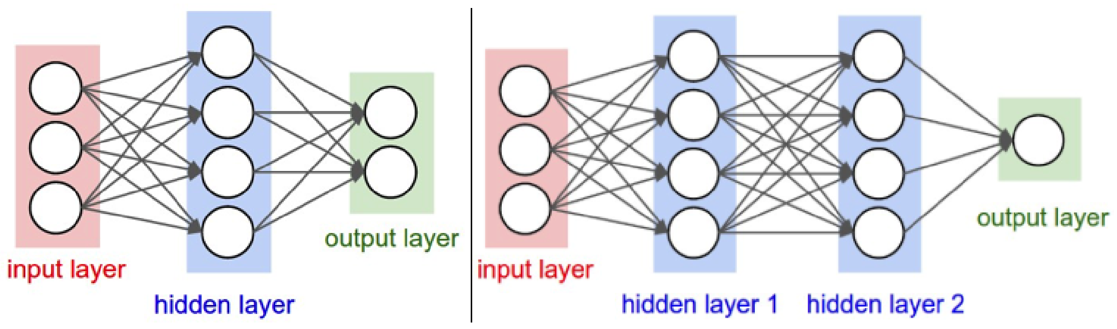


Figure 1.2.: Dense Neural Networks - Two Neural Networks consisting of dense layers. Left: 2 layers with one hidden layer, three inputs, and two outputs. Right: 3 layers with two hidden layers, three inputs, and one output. (Image source: [1])

accuracy.

While this universal approximation property sounds powerful, it is somewhat limited in practical applications. The fact that a two-layer neural network can approximate any function is not particularly useful. For instance, a simple function like $g(x) = \sum c_i \mathbb{1}(a_i < x < b_i)$, which is just a sum of indicator functions, is also a universal approximator, but it is not used in machine learning because it is not efficient for real-world data. [1]

1.4. Optimization

In the previous sections, we introduced the structure of neural networks and frequently referred to the weights of neurons, which determine the output of a neuron. But how do we find the correct weights for the neurons in the network so that they can perform a specific task or approximate a particular function? In other words, how do we "learn" the weights?

The following section aims to answer this question and explain how we can use gradients to optimize a network's weights to represent a specific function.

We will focus exclusively on "supervised learning", a data-driven approach where we optimize the weights of a network using training examples. For this, we have a training dataset of size N, consisting of tuples $(\mathbf{x}_i; \mathbf{y}_i)$. Here, $\mathbf{x}_i$ is the input to the network, such as an image, and $\mathbf{y}_i$ is the correct output corresponding to the input, such as the object in the image, like a cat or a dog. $\mathbf{y}_i$ is also referred to as the "label" or "ground truth". [1]

1.4.1. Gradient Based Optimization

First, we need to introduce the concept of the "loss function" v. It serves as a metric for evaluating the quality of a given set of parameters by assessing how closely the predicted values align with the ground truth labels in the training data. $\mathcal{L}$ depends on the network's output $\tilde{\mathbf{y}}_i$ and the label $\mathbf{y}_i$ and returns a scalar value: $\mathcal{L}(\tilde{\mathbf{y}}_i, \mathbf{y}_i) \in \mathbb{R}$. Since $\tilde{\mathbf{y}}_i(\mathbf{w})$ depends on the network's weights $\mathbf{w}$ and $(\mathbf{x}_i; \mathbf{y}_i)$ is fixed for a training example i, $\mathcal{L}$ depends only on the weights of the network: $\mathcal{L}(\mathbf{w})$. [1]

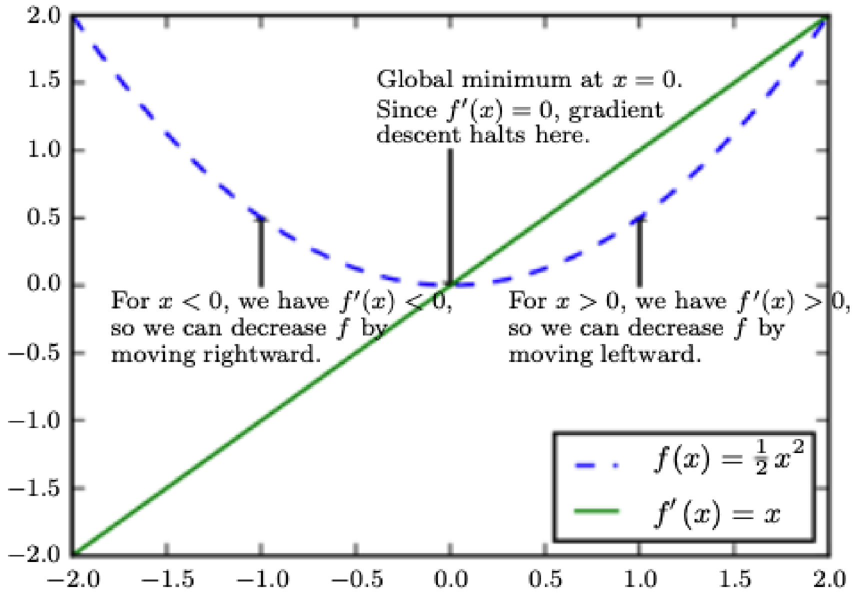


Figure 1.3.: Gradient Descent Visualization in 1D - In 1D, the gradient simplifies to the regular derivative, denoted as $\frac{d}{dx}$. (Image source: [2])

Our goal is to minimize the loss function by adjusting the weights of the network. One way to do this is by taking a step in the direction of the steepest descent of the function $\mathcal{L}$. This direction is given by the negative gradient. We can update our weights according to the following rule:

$$\mathbf{w}' = \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \tag{1.5}$$

where η is called the "learning rate", and it determines the step size because the gradient only tells us the direction of the steepest descent and not how far along we should go. To minimize $\mathcal{L}$, we repeatedly apply this update rule for different training examples. The procedure converges when $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \approx 0$. Figure 1.3 illustrates an example in the 1D case. [2]

1.4.2. Stochastic Gradient Descent

The loss function from 1.4.1 calculates the loss for a single training example. However, it is common to use a sum over many different training examples:

$$\mathcal{J}(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\tilde{\mathbf{y}}_i(\mathbf{w}), \mathbf{y}_i). \tag{1.6}$$

When the sum includes the entire training dataset, meaning $m = N$, the computation can often be too costly due to the large size of the dataset. Therefore, a subset $\mathbb{B} = \{(\mathbf{x}_1; \mathbf{y}_1), ..., (\mathbf{x}_m; \mathbf{y}_m)\}$ with $|\mathbb{B}| = m < N$ is randomly selected. This subset is called a "minibatch", and $|\mathbb{B}| = m$ is referred to as the batch size. The update rule is thus rewritten as:

$$\mathbf{w}' = \mathbf{w} - \eta \nabla_{\mathbf{w}} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\tilde{\mathbf{y}}_i(\mathbf{w}), \mathbf{y}_i) \tag{1.7}$$

[2]

1.4.3. Adam Optimizer

The Adam optimizer is widely regarded as one of the most effective and commonly used optimizers for deep learning models due to its adaptive nature and strong convergence properties. Unlike traditional optimization methods with a fixed learning rate, Adam adjusts the learning rate for each parameter, addressing issues encountered with simpler methods like vanilla gradient descent (shown in the update rule in equation 1.5).

Optimization methods that use a fixed learning rate, such as vanilla gradient descent, often struggle with certain difficulties during training. One common issue is that the learning rate remains constant across all parameters and throughout the entire training process. While this can work in simple cases, it presents problems when the optimizer encounters complex surfaces in the loss landscape, such as saddle points. In these scenarios, gradient descent may struggle to make meaningful progress, getting stuck in regions where gradients are small or oscillating chaotically.

To visualize these challenges, consider the case of a saddle point, where one dimension of the loss function curves up and another curves down. In this situation, a fixed learning rate can cause the parameter updates to stagnate because the gradients become very small in certain directions. This is where adaptive optimizers like Adam shine, as they adjust the learning rate based on the history of gradients for each parameter, allowing the model to escape such problematic areas more easily.

The Adam optimizer introduces two main components:

1. **Momentum**: A moving average of past gradients is used to smooth the updates, reducing noise in the gradient estimates.
2. **Adaptive Learning Rate**: Adam adapts the learning rate for each parameter based on the magnitude of its gradient, helping to balance progress across parameters that experience different gradient scales.

Let $\nabla\mathcal{L}$ be the gradient of the loss function and $\mathbf{w}$ the weights of the model. The Adam update rule can be simplified as follows:

$$m' = \beta_1 \cdot m + (1-\beta_1)\cdot\nabla\mathcal{L} \tag{1.8}$$

$$v' = \beta_2 \cdot v + (1-\beta_2)\cdot(\nabla\mathcal{L})^2 \tag{1.9}$$

$$\mathbf{w}' = \mathbf{w} - \eta\cdot\frac{m'}{\sqrt{v'}+\epsilon}. \tag{1.10}$$

Here, m represents the momentum term (first moment), v is the second moment (the moving average of squared gradients), and ϵ is a small constant added to avoid division by zero. The key idea is that Adam keeps track of both the gradient mean and variance, enabling it to adapt the learning rate for each parameter over time.

To further improve the convergence in the early stages of training, Adam includes a bias correction mechanism to adjust for the fact that both m and v are initialized to zero. This correction ensures that the optimizer does not underestimate the updates when the model has only seen a few gradients. The full update, including bias correction and the iteration number t, is:

$$m_{t+1} = \frac{\beta_1\cdot m_t + (1-\beta_1)\cdot\nabla\mathcal{L}}{1-\beta^t} \tag{1.11}$$

$$v_{t+1} = \frac{\beta_2\cdot v_t + (1-\beta_2)\cdot(\nabla\mathcal{L})^2}{1-\beta^t} \tag{1.12}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\cdot\frac{m_{t+1}}{\sqrt{v_{t+1}}+\epsilon}. \tag{1.13}$$

[1][3]

1.5. Backpropagation

Backpropagation is a fundamental algorithm used in training feedforward neural networks to calculate the gradient. When a neural network processes an input x to produce an output $\tilde{y}$, the information flows through the network layer by layer, a process known as forward propagation. During this process, the input propagates through hidden units in each layer until a final output is produced. The backpropagation algorithm enables the flow of information to move in the opposite direction, from the output back through the network. This reverse flow calculates the gradient of the scalar loss function $\mathcal{L}$, which is essential for updating the model's parameters during training.

Backpropagation uses the chain rule of calculus to compute derivatives efficiently in neural networks. The chain rule is used to determine the derivatives of functions that are composed of other functions as long as the derivatives of the individual functions are known.

Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, $f : \mathbb{R}^n \mapsto \mathbb{R}$ and $g : \mathbb{R}^m \mapsto \mathbb{R}^n$. If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \tag{1.14}$$

which can be written in vector notation as

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z, \tag{1.15}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g.

In backpropagation, the gradient of a variable $\mathbf{x}$ can be computed by multiplying a Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by a gradient $\nabla_{\mathbf{y}} z$. The backpropagation algorithm is essentially a sequence of these Jacobian-gradient products applied to each operation in the computational graph. By recursively applying the chain rule, one can compute the gradient for each node and parameter in the neural network graph.

1.6. Initialization

Before one can start the training of a neural network, one has to initialize the weights of the network. Since one does not know what the final weights should be, it is a good assumption that half of them are less than zero and the other half are greater than 0. However, if all weights were initialized with 0, there would be no asymmetry between the neurons. As a result, all neurons would produce the same output, calculate the same gradient during backpropagation, and receive the same update, which means that the neurons would not learn different features of the data. However, based on the above argument, you still want to initialize the weights as close to zero as possible. One possibility is to initialize the weights randomly as very small values. A multi-dimensional Gaussian or uniform distribution is particularly suitable for this [1].

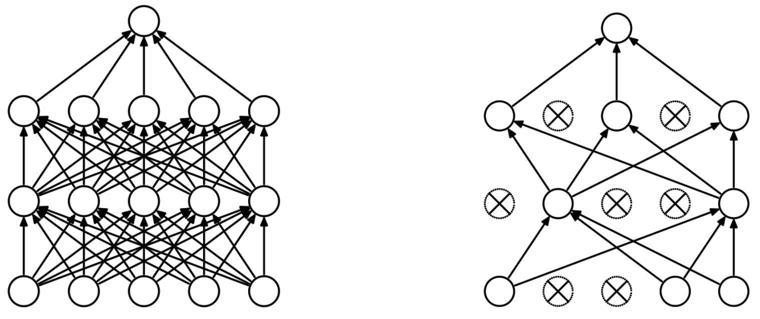
1.7. Dropout

Deep neural networks (DNN) are capable of modeling complex relationships between inputs and outputs through multiple non-linear hidden layers. However, this expressiveness comes at a cost: with limited training data, DNNs are prone to overfitting. Overfitting occurs when a model captures noise in the training data, leading to poor generalization when encountering new, unseen data. Dropout is a way to address overfitting.

In theory, the most effective way to regularize a fixed-sized model is to average the predictions of all possible parameter configurations, weighted by their posterior probabilities. This approach, while theoretically optimal, is computationally infeasible for large models due to the vast number of possible configurations.

Combining different models nearly always improves the performance of machine learning methods. In reality, one can often not use this approach because tuning the hyperparameters and training each individual neural network is a costly task.

Dropout is a regularization technique to address overfitting by combining many different networks in a computationally efficient way. The core idea behind dropout is to randomly drop neurons, along with their connections, during training. By doing so, dropout prevents the units from co-adapting too much, which forces each neuron to develop more robust features that are useful on their own (figure 1.4).

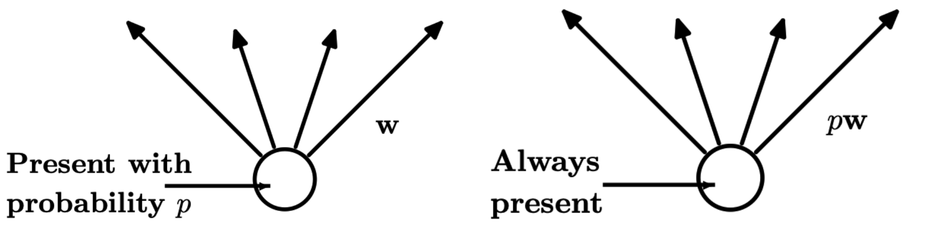

(a) Standard Neural Network

(b) Applying Dropout to the layers

Figure 1.4.: Standard NN & Dropout NN - Standard neural network (left) and network with applied dropout (right). Crossed neurons have been dropped. (Image source: [4])

During training, dropout draws samples from an exponentially large number of different "thinned" networks. Each is a subset of the original network with randomly selected neurons removed. This means that each training case is presented with a slightly different model. For a neural network with n neurons, this means the models during training are sampled from 2^n possible thinned neural networks.

At test time, instead of averaging the predictions from all these thinned networks, dropout



(a) Neuron is used with probability p during training (b) While testing the neuron is always used and its weights are multiplied by p

Figure 1.5.: Dropout: Training vs. Testing (Image source: [4])

uses a single, "unthinned" network with weights scaled down by the dropout probability (figure 1.5). This scaling ensures that the output at test time is an approximate geometric mean of the outputs from the exponentially many thinned networks trained during the training phase.

Consider a neural network comprising L hidden layers. Each hidden layer is indexed by $l \in \{1, \ldots, L\}$. The vector of inputs to layer l is represented by $\mathbf{z}^{(l)}$, while $y^{(l)}$ denotes the vector of outputs from layer l (with $\mathbf{y}^{(0)} = \mathbf{x}$ being the input of the network). The weights and biases at layer l are denoted by $W^{(l)}$ and $\mathbf{b}^{(l)}$.

The feed-forward operation in a standard neural network (figure 1.6 a) can be expressed as follows for $l \in \{0, \ldots, L-1\}$ and any hidden unit i:

$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)}\mathbf{y}^{(l)} + b_i^{(l+1)} \tag{1.16}$$

$$y_i^{(l+1)} = f(z_i^{(l+1)}) \tag{1.17}$$

where f is any activation function.

With dropout the feed-forward operation changes to (figure 1.6 b):

$$r_j^{(l)} \sim \text{Bernoulli(p)} \tag{1.18}$$

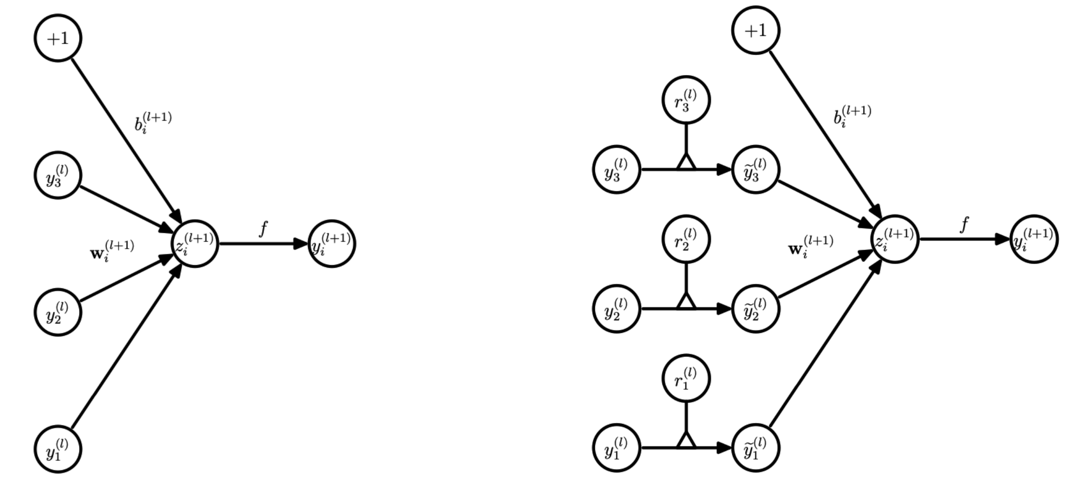
$$\widetilde{\mathbf{y}}^{(l)} = \mathbf{r}^{(l)} * \mathbf{y}^{(l)} \tag{1.19}$$

$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)}\widetilde{\mathbf{y}}^{(l)} + b_i^{(l+1)} \tag{1.20}$$

$$y_i^{(l+1)} = f(z_i^{(l+1)}) \tag{1.21}$$

$*$ denotes an element-wise product. For each layer l, $\mathbf{r}_j^{(l)}$ represents a vector of independent Bernoulli random variables, where each element has a probability p of being 1. This vector is sampled and applied element-wise to the layer's outputs $y^{(l)}$ via multiplication, yielding the thinned outputs $\tilde{y}^{(l)}$. These thinned outputs then become the inputs to the next layer. This process is repeated for each layer, effectively sampling a sub-network from the larger network.

During training, the loss function's derivatives are backpropagated through this sub-network. At test time, the weights are scaled as $W_{\text{test}}^{(l)} = pW^{(l)}$ and the neural network operates without dropout. [4]



(a) Standard Network Architecture

(b) Dropout Network Architecture

Figure 1.6.: Architecture of Standard Network and Network with Dropout (Image source: [4])

1.8. Layer Normalization

Layer normalization is a technique designed to improve the training of deep neural networks by addressing some of the limitations of batch normalization. While batch normalization has proven effective in accelerating training and improving model performance, it relies on the statistics of mini-batches, which can be problematic in certain scenarios, especially when working with small batch sizes. Layer normalization works by normalizing the summed inputs to the neurons within each layer of a neural network, but unlike batch normalization, it does so independently for each training case. Specifically, it calculates the mean and variance for normalization across all neurons in a layer for a single training example rather than across a mini-batch.

Let a^l represent the vector of summed inputs to the neurons in the l-th hidden layer in a deep feed-forward neural network. The summed inputs for the i-th hidden unit in the l-th layer are computed as:

$$a_i^l = {w_i^l}^T h^l, \tag{1.22}$$

where w_i^l are the weights of the i-th hidden unit in the l-th layer and h^l the output of the $(l-1)$-th layer. The output h_i^{l+1} of the layer is then calculated as:

$$h_i^{l+1} = f(a_i^l + b_i^l), \tag{1.23}$$

where $f(\cdot)$ is an element-wise non-linear function and b_i^l is the scalar bias parameter.

One of the challenges in training deep networks is the "covariate shift" problem, where the distribution of inputs to a layer changes during training as the parameters of the previous layers are updated. This can slow down training since each layer must continuously adapt to the shifting input distribution.

Batch normalization was proposed to address this issue by normalizing the inputs to each hidden unit over a mini-batch of training data. Specifically, for the i-th summed input in the l-th layer, batch normalization normalizes the input using the mean μ_i^l and variance σ_i^l computed over the mini-batch:

$$\bar{a}_i^l = \frac{g_i^l}{\sigma_i^l}(a_i^l - \mu_i^l) \tag{1.24}$$

$$\mu_i^l = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}[a_i^l], \tag{1.25}$$

$$\sigma_i^l = \sqrt{\mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}[(a_i^l - \mu_i^l)^2]}, \tag{1.26}$$

where $\bar{a}_i^l$ is the normalized summed input, and g_i^l is a gain parameter that scales the normalized activation before the non-linear activation function.

Layer normalization fixes the mean and variance of the summed inputs within the layer and thus computes the normalization statistics over all hidden units in the same layer as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l, \tag{1.27}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2}, \tag{1.28}$$

where H is the number of hidden units in a layer, all hidden units share the same μ^l and σ^l. Layer normalization can thus also be used for a batch size of 1. [5]

1.9. ResNet Architecture

The ResNet architecture, with its residual learning and shortcut connections, serves as the basis for the model used in this thesis. Although ResNet was initially developed for image recognition, the underlying principles of its architecture, such as residual learning and the use of shortcut connections, are applicable across various domains of machine learning. Understanding these principles provides valuable insights that can be extended to the study of long-range interactions in more complex systems. The following section offers an overview of the general ideas and design of ResNet, illustrating concepts that are foundational to the broader discussions later in this thesis.

In recent years, deep convolutional neural networks (CNNs) have led to significant advancements in image classification and recognition tasks. One of the pivotal developments in this field is the introduction of the Residual Network (ResNet) architecture, which was proposed to address the challenges associated with training very deep networks. The primary innovation of ResNet lies in its residual learning framework, which mitigates the degradation problem often encountered as network depth increases.

As networks grow deeper, it becomes increasingly difficult to train them effectively. Traditional deep networks often face the degradation problem, where adding more layers to a sufficiently deep model leads to higher training error rather than improving performance. This issue is not necessarily due to overfitting but is rather a consequence of optimization difficulties in very deep networks.

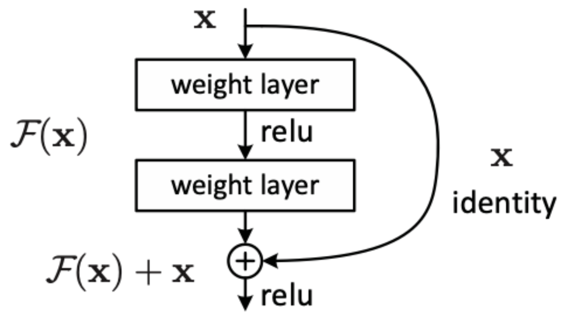

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1.7.: ResNet Block with "shortcut" connection (Image source: [7])

To address these challenges, ResNet introduces the concept of residual learning. Instead of each layer learning an unreferenced function, the ResNet layers are designed to learn residual functions with respect to the input layer. Formally, if the desired underlying mapping is denoted as $\mathcal{H}(x)$, ResNet reformulates the function to learn $\mathcal{F}(x) = \mathcal{H}(x) - x$, where the network learns the residual mapping $\mathcal{F}(x)$. The original function then becomes $\mathcal{F}(x) + x$.

This approach is based on the hypothesis that it is easier for the network to optimize the residual function $\mathcal{F}(x)$ than to directly approximate the desired mapping $\mathcal{H}(x)$. If the optimal mapping is indeed close to the identity mapping, then pushing the residual to zero (i.e., learning the identity function) becomes easier than fitting an identity mapping by stacking nonlinear layers.

The ResNet architecture is characterized by the use of ”shortcut connections” that skip one or more layers (see Figure 1.7). These shortcuts perform identity mappings and add the output of these mappings to the output of the stacked layers. This design does not introduce additional parameters or computational complexity, making the network easier to optimize while enabling the training of very deep models. This simple yet effective design has enabled the training of networks with unprecedented depth.

The introduction of ResNet has had a profound impact on the field of deep learning, particularly in image recognition tasks. The architecture has been shown to significantly outperform previous models, even when scaled to hundreds of layers. [7]

1.10. Kernel Ridge Regression

Kernel Ridge Regression (KRR) is a nonlinear regression method widely used in machine learning (ML) for interpolating data while applying regularization to avoid overfitting. This model is particularly relevant in molecular simulations, where accurate predictions of system properties are required without direct numerical solutions of complex equations, such as the Schrödinger equation in quantum systems. KRR provides a flexible approach to approximate complex, high-dimensional functions by learning from training data.

KRR is formulated as follows:

$$f^{\mathrm{ML}}(x) = \sum_{j=1}^{N_T} a_j k(x, x_j) \tag{1.29}$$

where $\{x_j\}$ for $j = 1, ..., N_T$ are the training data points, a_j are the learned weights, and $k(x, x_j)$ is a kernel function that measures the similarity between the input point x and the training points x_j.

The Gaussian kernel is commonly used in KRR defined as:

$$k(x, x') = \exp -\frac{(x - x')^2}{2\sigma^2} \tag{1.30}$$

where σ is the length scale hyperparameter controlling the degree of correlation between data points.

The weights a_j are found by minimizing the loss function

$$\mathcal{L}(\mathbf{a}) = \sum_{j=1}^{N_T} (f^{\mathrm{ML}}(x_j) - f_j) + \lambda \mathbf{a}^T \mathbf{K} \mathbf{a} \tag{1.31}$$

where $\mathbf{a} = (a_1, ..., a_{N_T})$ is the weight vector, f_j are the labels (or values to predict), $\mathbf{K}$ is the kernel matrix with elements $K_{ij} = k(x_i, x_j)$, and λ is a regularization parameter that controls the trade-off between fitting the training data and smoothness of the model.

The two hyperparameters, the length scale σ and regularization strength λ, play crucial roles in determining the model's performance but are not determined by equation 1.31.

σ controls how rapidly the kernel decays with distance. For small values of σ, the kernel becomes localized, and the model focuses on fitting individual points closely. For largeσ, the kernel becomes broader, leading to smoother fits across the training data.

λ determines how much the model is allowed to deviate from the training data. Small λ values force the model to fit the training data exactly, while large λ values allow the model to smooth out noise or irrelevant fluctuations in the data and reduce overfitting. [8]

1.11. Graph Neural Networks

Graph Neural Networks (GNNs) are a class of machine learning models designed to operate on graph-structured data, making them an ideal choice for molecular simulations, where molecules are naturally modeled as graphs. In molecular graphs, atoms are represented as nodes, while bonds between atoms serve as edges. This structural flexibility allows GNNs to capture the complex relationships within molecular systems, including both local interactions and long-range dependencies between atoms.

At their core, GNNs extend traditional deep learning methods to handle non-Euclidean data, such as graphs, where the relationships between entities are more complex than in grid-based structures like images. GNNs achieve this by learning node representations through a process called message passing or neighborhood aggregation.

1.11.1. Main Types of GNNs

Recurrent Graph Neural Networks (RecGNNs): These models iteratively propagate information between nodes until convergence. This was one of the first methods proposed for GNNs but tends to be computationally expensive due to the repeated message passing until a stable equilibrium is reached.

Convolutional Graph Neural Networks (ConvGNNs): Inspired by the success of CNNs in image processing, ConvGNNs define a convolution operation on graph data. In ConvGNNs, each node aggregates information from its neighbors through weighted combinations, much like a convolution filter operates on pixel neighborhoods in images. This method has been widely adopted in GNN applications due to its efficiency and scalability.

Graph Autoencoders (GAEs): These models focus on unsupervised learning, where graphs are encoded into a latent space and reconstructed, helping in tasks such as graph generation and node clustering.

Spatial-Temporal Graph Neural Networks (STGNNs): These models capture both spatial relationships in graphs and their dynamic changes over time, which can be useful in simulations where molecular structures change over time.

GNNs can be trained in a supervised, semi-supervised, or unsupervised manner, depending on the nature of the task.

1.11.2. GNNs in Molecular Simulations

In molecular simulations, GNNs are used for various tasks such as: Molecular property prediction: Predicting the chemical properties of molecules based on their graph structure. Learning molecular fingerprints: GNNs can infer molecular fingerprints from the graph representation, which is crucial for drug discovery and other chemical applications. Protein interface prediction: By treating protein structures as graphs, GNNs can predict interfaces between proteins, which is vital for understanding biological processes. [9]

2. Machine Learning for Molecular Simulation

As early as 1929, Paul Dirac pointed out that a large part of the physics of atoms and chemistry was already known. However, the equations were often too difficult to solve analytically. The challenge now was to find approximation methods that correctly described the properties of complex atomic systems without the need for many or complex calculations. Over 90 years later, this statement is still valid. Machine learning (ML) is one of the fastest-developing fields of our time, which means that it also has an impact on the approximation of complex atomic systems. As shown in detail in the previous chapter, ML allows us to find the underlying complex relationship between input and output data. This can also be very useful for atomic and molecular systems since the chemical properties depend on the atomic configuration. An ML algorithm can learn this relationship with enough data without solving equations [10].

All atomic simulations require a potential energy surface (PES) as input, which describes how the atoms interact with each other. The forces can then be derived from this. Solving the Schrödinger equation within the Born-Oppenheimer approximation is the most precise way to determine the PES. The most widely used method is the density function theory (DFT), which scales with N^3, where N is the number of atoms. For large systems with more than 1000 atoms and long simulation times of several nanoseconds, the computational cost is often very high. To overcome this problem, empirical potentials are often used, which describe a relationship between atomic positions and the energy of the system. Since many assumptions are involved in these relationships, these approaches are significantly less computationally intensive but also often less accurate. One thus finds oneself in a dilemma in that quantum mechanical methods require very precise but long computing times, and empirical potentials can be calculated more efficiently but are imprecise.[11]

2.1. Mathematical Modelling

The following section will present the mathematical modeling. The first part deals with Deep Potentials, which will also be used in the later approach. Additionally, the Atomic Cluster Expansion will be introduced as another approach. This has been included for clarity but will no longer play a role in the subsequent chapters.

2.1.1. Deep Potentials

Deep Potentials (DP) are neural network potentials (NNP) that describe the Potential Energy Surface (PES) of a system. In this context, consider a system consisting of N atoms. The total energy of the system is a function of the atomic coordinates $R = \{\mathbf{r_1}, \mathbf{r_2}, ..., \mathbf{r_i}, ..., \mathbf{r_N}\}$, where $\{r_{i1}, r_{i2}, r_{i3}\}$ are the three Cartesian components of the position vector $\mathbf{r_i}$ of atom i. The potential energy $E(R)$ can be accurately determined using first principle calculations. The force $\mathbf{F_i}$ ($\mathbf{F_i} = \{F_{i1}, F_{i2}, F_{i3}\}$) acting on atom i is the negative gradient of the potential energy with respect to its atomic coordinates, given by

$$\mathbf{F_i} = \nabla_{\mathbf{r_i}} E. \tag{2.1}$$

In many atomistic simulations, periodic boundary conditions are employed to model an infinite system using a finite number of atoms. These conditions are represented by cell vectors organized in a matrix $h = \{h_{\alpha\beta}\}$, where $h_{\alpha\beta}$ represents the β-th component of the α-th cell vector. The virial tensor $\Xi = \{\Xi_{\alpha\beta}\}$ is defined in terms of the derivative of the energy with respect to the cell vectors:

$$\Xi_{\alpha\beta} = -\frac{\partial E}{\partial h_{\gamma\beta}} h_{\gamma\beta}. \tag{2.2}$$

The process of training a machine learning potential (MLP) for this system falls under classical supervised learning. Initially, the total energy, atomic forces, and virial tensors of various system configurations are computed using first-principles methods. These computed values serve as training labels for the MLP. The potential is represented as $E^{\omega}(R)$, where ω denotes the set of trainable parameters of the model. The corresponding forces $\mathbf{F}^{\omega}(R)$ and virial tensors $\Xi^{\omega}(R)$ are derived from $E^{\omega}(R)$ using the equations 2.1 and 2.2. The training process involves minimizing a loss function that quantifies the difference between the predicted and true values of the energy, forces, and virials:

$$\mathcal{L} = \frac{1}{|\mathcal{B}|} \sum_{k \epsilon \mathcal{B}} (p_e \mathcal{L}_e^{(k)} + p_f \mathcal{L}_f^{(k)} + p_v \mathcal{L}_v^{(k)}) \tag{2.3}$$

$$\mathcal{L}_e^{(k)} = \frac{1}{N} |E(\mathcal{R}^{(k)}) - E^{\omega}(\mathcal{R}^{(k)}))|^2 \tag{2.4}$$

$$\mathcal{L}_f^{(k)} = \frac{1}{3N} \sum_{i\alpha} |F_{i\alpha}(\mathcal{R}^{(k)}) - F_{i\alpha}^{\omega}(\mathcal{R}^{(k)}))|^2 \tag{2.5}$$

$$\mathcal{L}_v^{(k)} = \frac{1}{9N} \sum_{i\alpha} |\Xi_{\alpha\beta}(\mathcal{R}^{(k)}) - \Xi^{\omega}_{\alpha\beta}(\mathcal{R}^{(k)})|^2 \tag{2.6}$$

$\mathcal{B}$ is a mini-batch of the training data, $|\mathcal{B}|$ is the number of configurations in this batch and p_e, p_f and p_v are the prefactors which are tunable Hyperparameters.

The extensibility of the total energy in MLPs is maintained by decomposing it into atomic contributions:

$$E^{\omega} = \sum_{i=1}^{N} E_i^{\omega} = \sum_{i=1}^{N} E_i^{\omega}(R_i) \tag{2.7}$$

where E_i^{ω} is the energy associated with atom i. We assume that the energy of an atom depends only on its coordinates and its local environment. This local environment is defined by a predefined cutoff radius r_c, within which the interactions between atoms are considered. The set of neighbor atoms within this cutoff distance is denoted as $\mathcal{N}_{r_c}(i) = \{j | r_{ij} = |\mathbf{r}_{ij}| \leqslant r_c\}$. The cardinality of this set is N_i, and the local environment of atom i is represented by the environment matrix $\mathcal{R}_i$, which has N_i rows and 3 columns, where each row j of R_i corresponds to the relative position of atom j with respect to atom i:

$$(\mathcal{R}_i)_j = (\mathbf{r}_{ij}). \tag{2.8}$$

The assumption that the atomic energy depends on the local environment is generally valid for systems with short-range interactions. However, long-range interactions, primarily arising from Coulombic forces within the electron density distribution, may also be significant in some materials. In metallic systems, shielding effects make the local environment assumption reasonable, as long-range interactions diminish rapidly with distance. Nevertheless, for materials where long-range interactions are dominant, they must be explicitly accounted for in the model. While various approaches have been proposed to include long-range interactions in MLPs, there is no universally accepted method to handle them appropriately. In the deep potential (DP) method, the focus is on systems where the local interaction assumption holds. For extended systems, periodic boundary conditions are applied to maintain consistency with the assumed local interactions.[11]

2.1.2. Atomic Cluster Expansion

The atomic cluster expansion (ACE) is a mathematical framework developed to provide an accurate and transferable description of the local atomic environment in molecular simulations. The primary goal of ACE is to offer a systematic and efficient method for representing the potential energy of a collection of atoms by expanding it in terms of local atomic clusters. This approach addresses the limitations of traditional many-atom expansions by ensuring rapid convergence and scalability, making it suitable for both small clusters and bulk materials.

The energy E of a system of N atoms can be expressed as a series expansion involving many-body interactions:

$$E = V_0 + \sum_i V^{(1)}(\mathbf{r}_i) + \frac{1}{2}\sum_{ij} V^{(2)}(\mathbf{r}_i, \mathbf{r}_j) + \frac{1}{3!}\sum_{ijk} V^{(3)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \cdots , \tag{2.9}$$

where $\mathbf{r}_i$ denotes the position of atom i, and $V^{(k)}$ represents the k-body interaction potential. The potentials are symmetric and uniquely defined, with higher-order terms diminishing in significance.

One can then separate the expansion into atomic contributions. The energy of atom i is represented as a sum over interactions with its neighbors:

$$E_i = V^{(1)}(\mathbf{r}_i) + \frac{1}{2}\sum_{j} V^{(2)}(\mathbf{r}_i, \mathbf{r}_j) + \frac{1}{6}\sum_{jk} V^{(3)}(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \cdots . \tag{2.10}$$

The convergence of the expansion is slow even if a cutoff radius r_c is introduced to take into account only atoms inside the cutoff radius. Evaluating up to the $(K+1)$-th term scales as N_c^K, where N_c represents the number of atoms inside the cutoff radius. The goal of ACE is to simplify the expansion for E_i.

E_i is completely defined by the $N-1$ vectors from atom i to all other atoms:

$$E_i(\sigma) = E_i(\mathbf{r}_{1i}, \mathbf{r}_{2i}, ..., \mathbf{r}_{Ni}). \tag{2.11}$$

$\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i$ and the collection of the $N-1$ vectors is abbreviated as $\sigma = (\mathbf{r}_{1i}, \mathbf{r}_{2i}, ..., \mathbf{r}_{Ni})$ since their order is irrelevant.

The next step is to define the inner product between two functions $f(\sigma)$ and $g(\sigma)$ as

$$\langle f|g\rangle = \int f^{\star}(\sigma) g(\sigma) d\sigma \tag{2.12}$$

where $f^{\star}(\sigma)$ denotes the complex conjugate of $f(\sigma)$.

Let $\phi_v(\mathbf{r})$ with $v = 0, 1, 2, \ldots$ be a set of orthogonal and complete basis functions that depend only on a single bond $\mathbf{r}$. For the set $\phi_v(\mathbf{r})$, the following holds:

$$\int \phi_v^{\star}(\mathbf{r}) \phi_u(\mathbf{r}) d\mathbf{r} = \delta_{vu} \tag{2.13}$$

$$\sum_{\nu} \phi_{\nu}^{\star}(\mathbf{r})\phi_{\nu}(\mathbf{r}') = \delta(\mathbf{r} - \mathbf{r}'). \tag{2.14}$$

A cluster α with K elements includes K bonds, denoted as $\alpha = (j_1 i, j_2 i, ..., j_K i)$, where the order of entries is irrelevant. The vector $\nu = (\nu_1, \nu_2, ..., \nu_K)$ represents the list of single-bond basis functions within the cluster. In ν, only single-bond basis functions with $v > 0$ are taken into account. Then, the cluster basis function is given by

$$\Phi_{\alpha\nu} = \phi_{\nu_1}(\mathbf{r}_{j_1 i})\phi_{\nu_2}(\mathbf{r}_{j_2 i})...\phi_{\nu_K}(\mathbf{r}_{j_K i}), \tag{2.15}$$

with $0 \leq K \leq N - 1$.

Since the orthogonality and completeness of the one-bond basis functions transfer to the cluster basis functions, it follows:

$$\langle \Phi_{\alpha\nu} | \Phi_{\beta\mu} \rangle = \delta_{\alpha\beta}\delta_{\nu\mu}, \tag{2.16}$$

$$1 + \sum_{\gamma \subseteq \alpha} \sum_{\nu} \Phi_{\gamma\nu}^{\star}(\sigma)\Phi_{\gamma\nu}(\sigma') = \delta(\sigma - \sigma'), \tag{2.17}$$

where α represents an arbitrary cluster, and the right-hand side of the completeness relation is the product of the corresponding right-hand sides of Eq. 2.14.

Let

$$k(\sigma, \sigma') = 1 + \sum_{\gamma\nu} \Phi_{\gamma\nu}^{\star}(\sigma)\Phi_{\gamma\nu}(\sigma') \tag{2.18}$$

be a kernel. One can then rewrite ACE of Eq. 2.10 into

$$E_i(\sigma) = \langle k(\sigma, \sigma') | E_i(\sigma') \rangle = J_0 + \sum_{\alpha\nu} J_{\alpha\nu}\Phi_{\alpha\nu}(\sigma). \tag{2.19}$$

The expansion coefficients $J_{\alpha\nu}$ can be derived through projection

$$J_{\alpha\nu} = \langle \Phi_{\alpha\nu} | E_i(\sigma) \rangle. \tag{2.20}$$

If atoms j and k are of the same chemical species, exchanging the bonds ji and ki does not alter the energy or any other atomic observable. Consequently, bonds within a cluster can be categorized by their chemical species. For a cluster with K bonds, where k_A, k_B, etc., represent the number of bonds to chemical species A, B, etc., the expansion coefficient $J_{\alpha\nu}$ is determined solely by the number of bonds to each species, denoted as $(J_{k_A k_B ... \nu})$. This principle simplifies the application of atomic cluster expansion to elements, as the expansion coefficient is fully characterized by the number of bonds within the cluster. Therefore, for elemental materials, the expansion coefficient can be expressed as

$$J_{\alpha\nu} = J_{\nu}^{(K)}. \tag{2.21}$$

One can then rewrite Eq. 2.19 into

$$\begin{aligned} E_i(\sigma) = & \sum_j \sum_\nu J_\nu^{(1)} \phi_\nu(\mathbf{r}_{ji}) \\ & + \frac{1}{2} \sum_{j_1 j_2}^{j_1 \neq j_2} \sum_{\nu_1 \nu_2} J_{\nu_1 \nu_2}^{(2)} \phi_{\nu_1}(\mathbf{r_{j_1 i}}) \phi_{\nu_2}(\mathbf{r}_{j_2 i}) \\ & + \frac{1}{3!} \sum_{j_1 j_2 j_3}^{j_1 \neq j_2, \ldots} \sum_{\nu_1 \nu_2 \nu_3} J_{\nu_1 \nu_2 \nu_3}^{(3)} \phi_{\nu_1}(\mathbf{r_{j_1 i}}) \phi_{\nu_2}(\mathbf{r}_{j_2 i}) \phi_{\nu_3}(\mathbf{r}_{j_3 i}) \\ & + \ldots, \end{aligned} \tag{2.22}$$

where J_0 was set to 0 and the coefficient $J_\nu^{(K)}$ contributes to the potential $V^{(K+1)}$.

The sum can be further transformed so that the summations become unrestricted,

$$\begin{aligned} E_i(\sigma) = & \sum_j \sum_\nu c_\nu^{(1)} \phi_\nu(\mathbf{r}_{ji}) \\ & + \frac{1}{2} \sum_{j_1 j_2} \sum_{\nu_1 \nu_2} c_{\nu_1 \nu_2}^{(2)} \phi_{\nu_1}(\mathbf{r_{j_1 i}}) \phi_{\nu_2}(\mathbf{r}_{j_2 i}) \\ & + \frac{1}{3!} \sum_{j_1 j_2 j_3} \sum_{\nu_1 \nu_2 \nu_3} c_{\nu_1 \nu_2 \nu_3}^{(3)} \phi_{\nu_1}(\mathbf{r_{j_1 i}}) \phi_{\nu_2}(\mathbf{r}_{j_2 i}) \phi_{\nu_3}(\mathbf{r}_{j_3 i}) \\ & + \ldots. \end{aligned} \tag{2.23}$$

Both equations are identical, allowing the coefficients $c_\nu^{(K)}$ to be calculated from $J_\nu^{(K)}$.

The costly N_c^K scaling can be avoided by a rearrangement of Eq. 2.23. Let the atomic basis be defined by projecting the basis functions onto the atomic density:

$$A_{i\nu} = \langle \rho_i | \phi_\nu \rangle = \sum_j \phi_\nu(\mathbf{r}_{ji}). \tag{2.24}$$

The atomic density ρ_i of an elemental material is defined as

$$\rho_i = \sum_j \delta(\mathbf{r} - \mathbf{r}_{ji}). \tag{2.25}$$

Substituting into Equation 2.23 gives

$$\begin{aligned} E_i(\sigma) = & \sum_\nu c_\nu^{(1)} A_{i\nu} \\ & + \sum_{\nu_1 \nu_2}^{\nu_1 \geqslant \nu_2} c_{\nu_1 \nu_2}^{(2)} A_{i\nu_1} A_{i\nu_2} \\ & + \sum_{\nu_1 \nu_2 \nu_3}^{\nu_1 \geqslant \nu_2 \geqslant \nu_3} c_{\nu_1 \nu_2 \nu_3}^{(3)} A_{i\nu_1} A_{i\nu_2} A_{i\nu_3} \\ & + \ldots. \end{aligned} \tag{2.26}$$

which is a polynomial in A_{iv}.

The construction of the atomic basis A_{iv} scales linearly with the number of neighbors N_c, while the energy evaluation, as given in Eq. 2.26, is independent of N_c. This implies that the time required to evaluate the energy scales linearly with the number of neighbors, regardless of the expansion order. Consequently, the atomic cluster expansion effectively addresses the poor scaling of the traditional many-atom expansion in Eq. 2.10 (see Figure 2.1). This improvement is crucial for the efficient evaluation of higher-order terms in densely packed materials with hundreds of atoms within the cutoff sphere.[12]

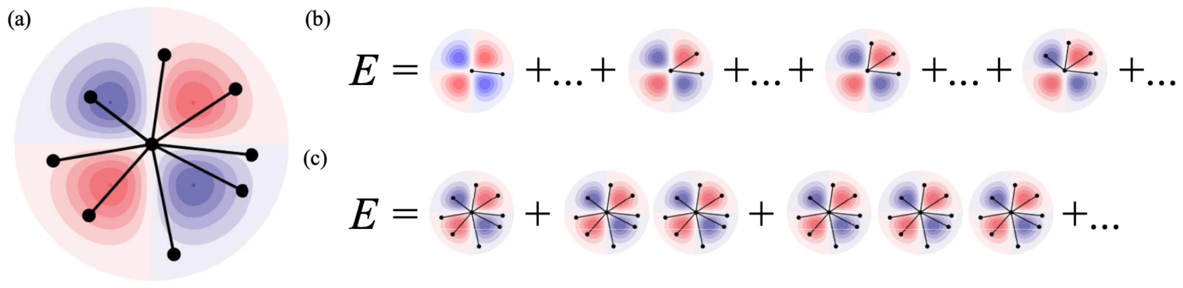


Figure 2.1.: Illustration of Atomic Cluster Expansion - (a) Basis function evaluated for the bonds between the central atom and its neighboring atoms (Eq. 2.10). (b) Atomic cluster expansion is expressed as a sum of pair, three-body, four-body, and higher-order contributions (Eq. 2.19). (c) Atomic Cluster expansion, which is a polynomial of the atomic base and scales linearly with the number of neighbors regardless of the expansion order (Eq. 2.26). (Image source: [12])

2.2. Symmetries

In molecular simulations, embedding known physical principles into machine learning (ML) models provides a substantial advantage over traditional ML tasks such as image classification. This approach ensures that the model's predictions remain physically plausible, thereby improving both accuracy and reliability.

For instance, in predicting the potential energy and atomic forces of the diatomic molecule O_2 in a vacuum, several physical rules must be adhered to:

1. Translational and Rotational Invariance: The energy of the molecule remains invariant under translation or rotation. Therefore, we can choose arbitrary positions for the atoms, reducing the energy to a function of the interatomic distance d only: $U(\mathbf{x}) \rightarrow U(d)$.
2. Conservation of Energy: The energy is conserved. The relationship between energy and force is given by $\mathbf{F}(\mathbf{x}) = -\nabla U(\mathbf{x})$, allowing us to compute force components accordingly.
3. Permutational Invariance: Exchanging the labels of identical atoms does not change the energy of the system.

[10]

2.3. Descriptors

In machine learning, two primary approaches are used to handle the invariances or symmetries from section 2.2: data augmentation (section 2.3.1) and incorporating the invariances directly into the ML model (section 2.3.2). [10]

2.3.1. Why Data Augmentation isn't sufficient

Data augmentation involves learning invariances by artificially generating additional training data and applying the known invariances to it. A given training point with positions as input and energy/force labels, $(\mathbf{x}; U, \mathbf{F})$, can be augmented by using the translational invariance of energy and force. This is done by adding more training data points $(\mathbf{x} + \delta\mathbf{x}; U, \mathbf{F})$ with random displacements $\delta\mathbf{x}$. Data augmentation enhances the robustness of the ML model and helps approximate the invariances, making it a valuable ML tool due to its ease of implementation. However, for some invariances, embedding them directly into the ML model can be conceptually challenging or computationally expensive.

Despite its benefits, data augmentation is statistically inefficient since it requires more training data, and it is also inaccurate because a network without built-in translation invariance will not predict constant energy when translating the molecule. This inaccuracy can result in unphysical predictions if one uses this energy model in an MD integrator. [10]

2.3.2. Implementing Descriptors

To maintain energy invariance, descriptors of the atomic environment are introduced that remain unchanged when the symmetry operations from section 2.2 are applied to the atomic coordinates:

$$\mathcal{D}(\mathcal{R}_i) = \mathcal{D}(U\mathcal{R}_i). \tag{2.27}$$

The atomic energy can thus be expressed as

$$E_i^w(\mathcal{R}_i) = \mathcal{F}(\mathcal{D}(\mathcal{R}_i)), \tag{2.28}$$

where $\mathcal{F}$ is the deep neural network. To calculate the atomic forces and the virial tensor using derivatives, the descriptor function $\mathcal{D}$ must be smooth.

There are two types of descriptors: non-smooth, as in [13], and smooth, as in [14], mappings of the atomic coordinates.[11]

Non-Smooth Descriptors

Non-smooth descriptors transform each atom i and its neighboring atoms $N_{r_c}(i)$, which lie within the cutoff radius r_c, into a local coordinate system and then sort them based on their distance to the central atom i. This transformation makes the atomic environment invariant to translation, rotation, and permutation.

To construct the local coordinate system for atom i, two neighboring atoms $a(i) \epsilon N_{r_c}(i)$ and $b(i) \epsilon N_{r_c}(i)$ are first selected from the local environment $N_{r_c}(i)$ such that atoms i, $a(i)$, and $b(i)$ are not colinear. In practice, $a(i)$ and $b(i)$ are often chosen as the two atoms nearest to i. The rotation matrix is defined as follows:

$$\mathcal{R}(\mathbf{r}_{ia(i)}, \mathbf{r}_{ib(i)}) = \begin{pmatrix} \mathbf{e}(\mathbf{r}_{ia(i)}) \\ \mathbf{e}\left[\mathbf{r}_{ib(i)} - \frac{\mathbf{r}_{ia(i)} \cdot \mathbf{r}_{ib(i)}}{\mathbf{r}_{ia(i)} \mathbf{r}_{ia(i)}} \mathbf{r}_{ia(i)}\right] \\ \mathbf{e}(\mathbf{r}_{ia(i)} \times \mathbf{r}_{ib(i)}) \end{pmatrix}^T , \tag{2.29}$$

where the elements in each column are the basis vectors in the local coordinate system, and $\mathbf{e}(\mathbf{r}) = \mathbf{r}/|\mathbf{r}|$ is the normalized vector of $\mathbf{r}$. The global coordinates $\mathbf{r}_{ij} = (x_{ij}, y_{ij}, z_{ij})$ are transformed into the local coordinates $\mathbf{r}'_{ij} = (x'_{ij}, y'_{ij}, z'_{ij})$ using the formula

$$(x'_{ij}, y'_{ij}, z'_{ij}) = (x_{ij}, y_{ij}, z_{ij}) \cdot \mathcal{R}(\mathbf{r}_{ia(i)}, \mathbf{r}_{ib(i)}), \tag{2.30}$$

where $\mathcal{R}(\mathbf{r}_{ia(i)}, \mathbf{r}_{ib(i)})$ is the rotation matrix.

From the local coordinates, the descriptor can now be constructed. The full information is obtained from

$$\{D_{ij}\} = \left\{ \frac{1}{r_{ij}}, \frac{x_{ij}}{r_{ij}}, \frac{y_{ij}}{r_{ij}}, \frac{z_{ij}}{r_{ij}} \right\}^{\text{sort}}_{j \epsilon N_{r_c}(i)} . \tag{2.31}$$

For atoms that are far from the central atom i, it may be sufficient to use only the radial information:

$$\{D_{ij}\} = \left\{ \frac{1}{r_{ij}} \right\}^{\text{sort}}_{j \epsilon N_{r_c}(i)} . \tag{2.32}$$

The superscript 'sort' indicates that the sorting is based on the inverse distance to atom i.

The advantage of non-smooth descriptors is that they retain complete information about the local environment. The descriptor is non-smooth because the selection of the neighboring atoms $a(i)$ and $b(i)$ is arbitrary. For example, if these are chosen as the two nearest neighbors, a continuous change in atomic positions can result in a discontinuous change in the local coordinate system and, thus, the local coordinates. Additionally, sorting is also not a continuous operation, resulting in additional discontinuities in the descriptor.[13]

Smooth Descriptors

The smooth descriptor will be briefly explained here, as it will be detailed and applied in section 4.

After constructing the environment matrix $\mathcal{R}_i \epsilon \mathbb{R}^{N_i \times 3}$ for atom i, it is transformed using the function $s(r_{ij})$, where s is a continuous and differentiable function:

$$s(r_{ji}) = \left\{ \begin{array}{ll} \frac{1}{r_{ji}}, & r_{ji} < r_{cs} \\ \frac{1}{r_{ji}} f_c(r_{ij}), & r_{cs} < r_{ji} < r_c \\ 0, & r_{ji} > r_c \end{array} \right\} \tag{2.33}$$

$f_c(r_{ij})$ decreases from 1 at r_{cs} to 0 at r_c. f_c should be differentiable to the second order. The extended matrix $\widetilde{\mathcal{R}}_i \epsilon \mathbb{R}^{N_i \times 4}$ is then obtained as follows:

$$(\widetilde{\mathcal{R}}_i)_j = s(r_{ij}) \times (1, \frac{x_{ij}}{r_{ij}}, \frac{y_{ij}}{r_{ij}}, \frac{z_{ij}}{r_{ij}}). \tag{2.34}$$

The embedding matrix $\mathcal{G}_i \epsilon \mathbb{R}^{N_i \times M}$ is then formed from the first column of the matrix $\widetilde{\mathcal{R}}_i$.

$$(\mathcal{G}_i)_j = (G_1(s(r_{ij}), Z_j), ..., G_M(s(r_{ij}), Z_j), \tag{2.35}$$

where $(G_1, ..., G_M)$ is a mapping of a deep neural network from the scalar input $s(r_{ij})$ and the atomic number Z_j of the j-th neighboring atom.

The smooth descriptor $\mathcal{D}_i$ is calculated from the extended matrix $\widetilde{\mathcal{R}}_i$ and the embedding matrix $\mathcal{G}_i$:

$$\mathcal{D}^i = \mathcal{G}^{i_1} \widetilde{\mathcal{R}}^i (\widetilde{\mathcal{R}}^i)^T \mathcal{G}^{i_2}, \tag{2.36}$$

where $i_1 = M$ and $i_2 < M$ are the first first i_2 columns of $\mathcal{G}^i$.

A smooth descriptor has the advantage that due to the function $s(r_{ij})$, the matrix only changes slightly with a small change in the position of a neighboring atom, preventing any abrupt jumps.[13]

3. Fast Multipole Method

The fast multipole method (FMM), summarized from Ref. [15], is a computational technique designed to efficiently solve the N-body problem, which involves calculating potential fields due to a distribution of sources. This method significantly reduces the computational complexity from $O(N^2)$ to $O(N)$, where N is the number of points. The FMM is applicable in various fields, including astrophysics, electrostatics, and wave scattering.

The essence of FMM lies in approximating the interactions between distant clusters of points rather than computing each interaction individually. There are two main categories of FMM based on the type of kernel functions: non-oscillatory and oscillatory kernels. This chapter explores the non-oscillatory type, providing a conceptual overview and detailing the classical FMM algorithm.

3.1. Theory

Let $X \subset \mathbb{R}^d$ be the N target points, $Y \subset \mathbb{R}^d$ the N source points, $G(x, y)$ a kernel function and $\{f(y) : y \epsilon Y\}$ a set of weights. One then wants to compute the potential $u(x)$ for each $x \epsilon X$ defined by

$$u(x) = \sum_{y \epsilon Y} G(x, y) f(y). \tag{3.1}$$

In the following, $X = Y = P$ is assumed.

For non-oscillatory kernels, such as the Coulomb potential $G(x, y) = \frac{1}{|x-y|}$, the FMM efficiently computes interactions between points distributed quasi-uniformly within a unit box (see Figure 3.1). The classical FMM uses a hierarchical decomposition of the computational domain into a quadtree structure (see Figure 3.3).

Before starting, let us first consider two disjoint squares, A and B, each containing $O(n)$ points. When A and B are well-separated, we can approximate the interactions between all points in A and B using a simplified procedure. E.g., imagine A and B as two distant galaxies. Instead of computing all pairwise interactions, we sum the masses in B to obtain a single equivalent mass $f_B = \sum_{y \epsilon B \cap P}$ at B's center c_B. The potential $u_A = G(c_A, C_B) f_B$ at A's center c_A is then computed and used to approximate the potential $u(x) = u_A$ at all points $x \epsilon X$ in A. This three-step procedure is efficient, reducing the computational cost from $O(n^2)$ to $O(n)$ (see Figure 3.2).

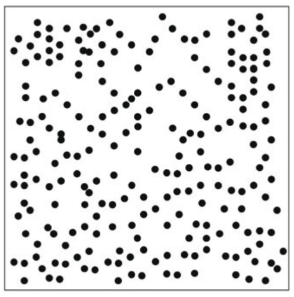


Figure 3.1.: N points quasi-uniformly distributed in a unit box (Image source: [15])

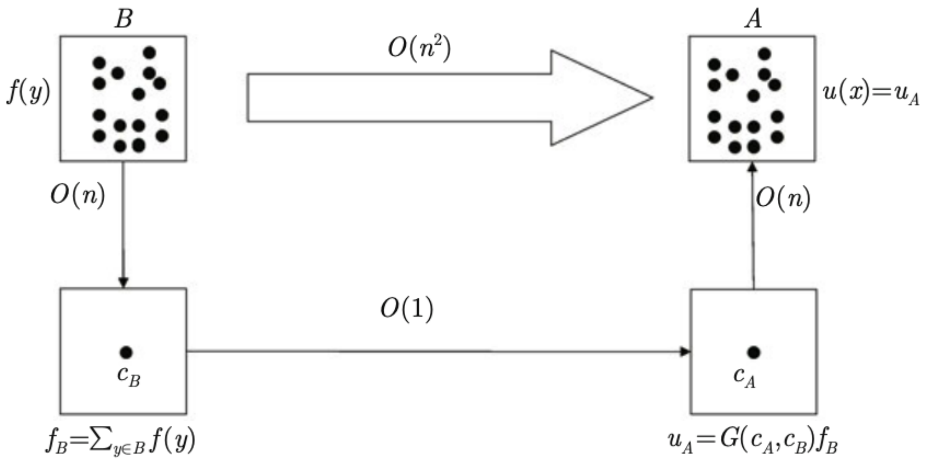


Figure 3.2.: FFM three-step procedure - A three-step procedure efficiently approximates the potential in region A due to sources in region B reducing the computational cost from $O(n^2)$ to $O(n)$ (Image source: [15]).

Only the potential in A from points in B were considered in this simple scenario. However, our goal is to analyze interactions among all points, so the three-step procedure is only a partial solution. To address this, one hierarchically partitions the domain using a quadtree structure until each leaf box contains fewer than a predetermined $O(1)$ number of points (see Figure 3.3). The entire quadtree consists of $O(\log N)$ levels, with the top level labeled as level 0. At level ℓ, there are 4^ℓ squares, each containing $O(N/4^\ell)$ points due to the quasi-uniform point distribution.

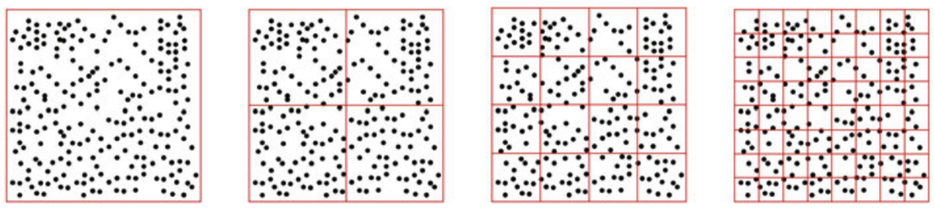


Figure 3.3.: FFM Quadtree Structure - The domain is divided using a quadtree structure until each leaf node contains a limited number of points, bounded by a small constant (Image source: [15]).

The algorithm begins at level 2 (see Figure 3.4 top-left). Let B be an arbitrary box at this level. B's near field, $N(B)$, includes B and its neighboring boxes, while the far field, $F(B)$, consists of all other boxes (complement of the near field). The interaction list for B is defined as the set of boxes in its far field. There are 4^2 possibilities for B, and for each B, there are $O(1)$ choices for A (see Figure 3.4 top-left). A and B contain $O(N/4^2)$ points due to the quasi-uniform distribution assumption. Thus, the costs on this level are

$$4^2 \cdot O(1) \cdot O(N/4^2) = O(N). \tag{3.2}$$

The interaction between B and its near field has to be calculated on the next level. Let B be a box on the next level (see Figure 3.4 top-right). We can ignore B's interaction with the far field of its parent since the previous level already handled it. Thus, we only need to account for interactions between B and the near field of its parent. At this level, there are 6^2 boxes in the parent's near field. Typically, 27 of these are well-separated from A. B's interaction list consists of these boxes. The cost at this level is

$$4^3 \cdot O(1) \cdot O(N/4^3) = O(N) \tag{3.3}$$

because each box at this level contains $O(N/4^3)$ points and there are 4^3 possibilities for B.

For the interactions between this B and its near field, we have to move down again one level. At a general level ℓ, there are 4^ℓ possible choices for B (see Figure 3.4, bottom-left), and for each B, there are at most 27 possible choices for A with each box at this level containing $O(N/4^\ell)$ points. Thus, the cost of the far-field computation is

$$4^\ell \cdot O(1) \cdot O(N/4^\ell) = O(N). \tag{3.4}$$

Upon reaching the leaf level, interactions between a leaf box B and its neighbors still need to be considered (see Figure 3.4 bottom-right). For these interactions, we use direct computation. Given that there are $O(N)$ leaf boxes, each containing $O(1)$ points and having $O(1)$ neighboring boxes, the overall cost of direct computation is

$$O(N) \cdot O(1) \cdot O(1) = O(N). \tag{3.5}$$

Thus, the cost at each level is $O(N)$, and there are $O(\log N)$ levels, the cost of performing the whole algorithm is $O(N \log N)$.

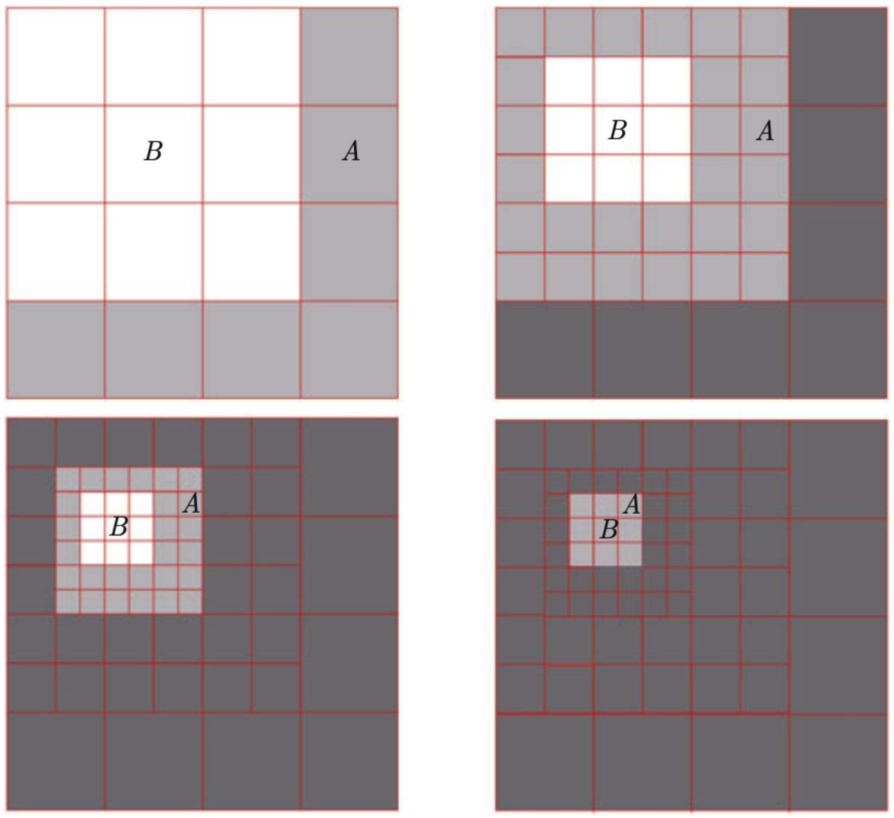


Figure 3.4.: FFM algorithm at different levels. Dark-gray: boxes for which the interaction already has been considered. Light-gray: Interactions considered at the current level.(Image source: [15]).

To improve this algorithm, we notice that

$$f_B = f_{B_1} + f_{B_2} + f_{B_3} + f_{B_4}, \tag{3.6}$$

because $B = B_1 \cup B_2 \cup B_3 \cup B_4$ and all of B's children B_i being disjoint. Therefore, assuming f_{B_i} are prepared, computing f_B using the previous line is significantly more efficient than summing over all $f(y)$ in B (see Figure 3.5 left). Likewise, for each A, we update $u(x)$ as

$u(x) := u(x) + u_A$ for all $x \in A$. Since we perform the same update for each children A_i ($i \in \{1, ..., 4\}$) and each x belongs to one of these A_i, we can just more efficiently update $u_{A_i} := u_{A_i} + u_A$ instead (see Figure 3.5 right). This means for f_b, we have to visit the parent after the children, and for u_A, the children after the parent (see Figure 3.5). [15]

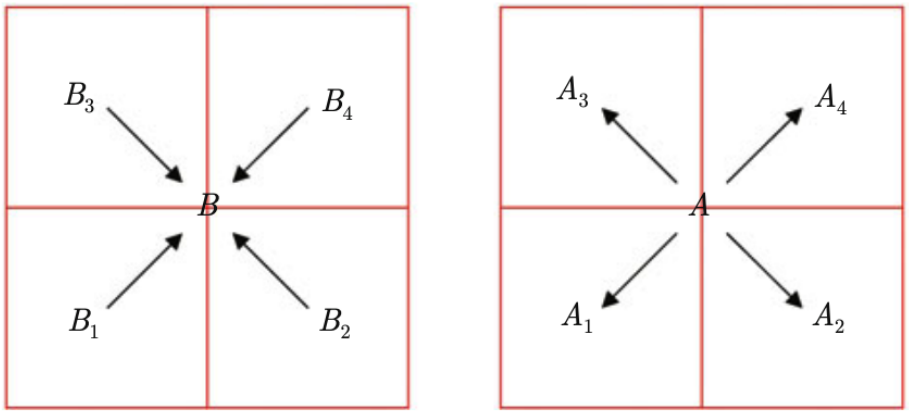


Figure 3.5.: Computation of f_B and u_A - Left: Directly compute f_B from f_{B_i}. Right: Add u_{A_i} to A's children.(Image source: [15]).

3.2. Algorithm

Both improvements result in the following final algorithm with a total cost of $O(N)$:

1. Initialization:
   For each level ℓ and each box A on level ℓ, set u_A to be zero.
2. Upward Pass:
   For levels from the leaf level $L-1$ to level 0:
   - If B is a leaf box, compute $f_B = \sum_{y \epsilon B \cap P} = f(y)$.
   - If B is not a leaf box, compute f_B as the sum of the equivalent sources from its child boxes: $f_B = f_{B_1} + f_{B_2} + f_{B_3} + f_{B_4}$.
3. Interaction Computation:
   For each level ℓ, for each box B on level ℓ, and for each box A in B's interaction list, update $u_A := u_A + G(c_A, c_B) f_B$
4. Downward Pass:
   For level 0 to level $L-1$ and for each box A
   - If A is a leaf box, update $u(x) := u(x) + u_A$.
   - If A is not a leaf box, update $u_{A_i} := u_{A_i} + u_A$ for each child A_i of A.
5. Direct Computation for Near Field:
   For each box B on the leaf level, update the potential: $u(x) := u(x) + \sum_{y \epsilon N(B) \cap P} G(x,y) f(y)$

The cost at steps 1 to 5 is $O(N)$ because there are at most $O(N)$ boxes in the tree, and the cost per box is $O(1)$, resulting in a total cost of $O(N)$. [15]

4. Multipole Method for Long Range Interaction in Neural Networks

Long-range interactions are critical to accurately modeling molecular systems, especially when dealing with potential energy surfaces. These interactions, which span beyond immediate neighbors, play a pivotal role in defining the global behavior of the system. However, including long-range interactions in machine learning models presents a significant computational challenge due to the quadratic scaling of pairwise distance calculations between atoms.

The baseline approach from [14] focuses on efficiently representing local environments around each atom within a cut-off radius by constructing symmetry-preserving descriptors. This reduces the computational complexity since only near neighbors are considered. Despite its effectiveness for short-range interactions, it does not capture the necessary long-range effects, which are crucial for the precise modeling of molecular systems.

To incorporate long-range interactions while maintaining computational efficiency, one can use an extension to the baseline model inspired by the Fast Multipole Method (FMM). This extension introduces a grid-based method, where the molecular system is divided into cells, and interactions between distant atoms are approximated by considering the center of mass of the grid cells. This approach significantly reduces the computational complexity by focusing on regions of space rather than individual atoms. Furthermore, this method ensures that the long-range effects are accounted for without needing to calculate every pairwise interaction beyond the cutoff radius, providing a scalable solution to modeling complex molecular systems.

This chapter will show how this method is implemented, focusing on how it integrates with the existing neural network framework to efficiently account for long-range interactions.

4.1. Baseline Model

To accurately and efficiently model potential energy surfaces while maintaining the necessary symmetries, it is essential to construct functions that inherently preserve these symmetries by introducing a descriptor. To effectively represent a scalar function $f(\mathbf{r})$ that is invariant under translation $\widehat{T}_{\mathbf{b}}f(\mathbf{r}) = f(\mathbf{r}+\mathbf{b})$, rotation $\widehat{R}_{\mathcal{U}}f(\mathbf{r}) = f(\mathbf{r}\mathcal{U})$ and permutation $\widehat{P}_{\mathcal{U}}f(\mathbf{r}) = f(\mathbf{r}_{\sigma(1)}, \mathbf{r}_{\sigma(2)}, ..., \mathbf{r}_{\sigma(N)})$, where $\mathbf{b}\epsilon\mathbb{R}^3$ is an arbitrary 3-dimensional translation vector, $\mathcal{U}\epsilon\mathbb{R}^{3\times 3}$ is an orthogonal rotation matrix, and σ denotes an arbitrary permutation of the set indices.

Neural networks have the capacity to fit various functions, and the key to ensuring that our representation preserves symmetry is to map the original input $\mathbf{r}$ into symmetry-preserving components. These components must accurately reflect the input $\mathbf{r}$ up to a symmetry operation.

To achieve translational and rotational invariance, one transforms each local environment matrix $\mathcal{R}^i$ in the following way:

$$\Omega^i \equiv \mathcal{R}^i(\mathcal{R}^i)^T, \tag{4.1}$$

which contains all necessary information about the point pattern around atom i and is invariant under translation and rotation. However, this matrix is not invariant under permutation.

Any permutation-invariant function $f(\mathbf{r})$ can be represented as $\rho(\sum_i \phi(\mathbf{r}_i))$, where $\phi(\mathbf{r}_i)$ is a multidimensional function and ρ is another general function.

For example,

$$\sum_i g(\mathbf{r}_i)\mathbf{r}_i \tag{4.2}$$

is invariant under permutation for any scalar function g.

The approach starts with constructing the local environment matrix for each atom i of the molecule. Let $\mathcal{N}_{r_c}(i)$ be the neighboring atoms inside the cutoff radius r_c of atom i and N_i the cardinality of this set (see section 2.1.1). The local environment matrix $\mathcal{R}^i\epsilon\mathbb{R}^{N_i\times 3}$ is defined as

$$\mathcal{R}^i = \{\mathbf{r}_{1i}^T, ..., \mathbf{r}_{ji}^T, ..., \mathbf{r}_{N_i i}^T\}, \mathbf{r}_{ji} = (x_{ji}, y_{ji}, z_{ji}) \tag{4.3}$$

and is constructed by defining the relative coordinates $\mathbf{r}_{ji} \equiv \mathbf{r}_j - \mathbf{r}_i$ where j $(1 \leq j \leq N_i)$ is the index of the j-th neighbor of atom i. The distance between atom i and its j-th neighbor is $r_{ji} = \|\mathbf{r}_{ji}\|$.

After constructing the local environment matrix $\mathcal{R}^i$ it is mapped onto generalized coordinates $\widetilde{\mathcal{R}}^i\epsilon\mathbb{R}^{N_i\times 4}$ by transforming each row $\{x_{ji}, y_{ji}, z_{ji}\}$ of $\mathcal{R}^i$ into a row of $\widetilde{\mathcal{R}}^i$:

$$\{x_{ji}, y_{ji}, z_{ji}\} \mapsto \{s(r_{ji}), \hat{x}_{ji}, \hat{y}_{ji}, \hat{z}_{ji}\}. \tag{4.4}$$

The components are defined as follows $\hat{x}_{ji} = \frac{s(r_{ji})x_{ji}}{r_{ji}}$, $\hat{y}_{ji} = \frac{s(r_{ji})y_{ji}}{r_{ji}}$ and $\hat{z}_{ji} = \frac{s(r_{ji})z_{ji}}{r_{ji}}$.

$s(r_{ji}) : \mathbb{R} \mapsto \mathbb{R}$ is a continuous and differentiable scalar weighting function:

$$s(r_{ji}) = \left\{ \begin{array}{ll} \frac{1}{r_{ji}}, & r_{ji} < r_{cs} \\ \frac{1}{r_{ji}}\{\frac{1}{2}\cos[\pi\frac{(r_{ji}-r_{cs})}{(r_c-r_{cs})}] + \frac{1}{2}\}, & r_{cs} < r_{ji} < r_c \\ 0, & r_{ji} > r_c \end{array} \right\} \tag{4.5}$$

where r_{cs} ensures that the components of $\widetilde{\mathcal{R}}^i$ smoothly go to zero at the boundary of the local region defined by r_c and remove its discontinuity there.

$$\widetilde{\mathcal{R}}^i = \begin{pmatrix} s(r_{1i}) & \hat{x}_{1i} & \hat{y}_{1i} & \hat{z}_{1i} \\ ... & ... & ... & ... \\ s(r_{ji}) & \hat{x}_{ji} & \hat{y}_{ji} & \hat{z}_{ji} \\ ... & ... & ... & ... \\ s(r_{N_i i}) & \hat{x}_{N_i i} & \hat{y}_{N_i i} & \hat{z}_{N_i i} \end{pmatrix}, \tag{4.6}$$

Let α_j and α_i be the chemical species of atom j and i. Then $G^{\alpha_j,\alpha_i}(s(r_{ji}))$ is defined as the local embedding network mapping $s(r_{ji})$ to M_1 outputs. The network parameters depend on the chemical species of atoms j and i. The local embedding matrix $\mathcal{G}^i \epsilon \mathbb{R}^{N_i \times M_1}$ is defined as:

$$(\mathcal{G}^i)_{jk} = (G(s(r_{ji})))_k. \tag{4.7}$$

Finally, the encoded feature matrix $\mathcal{D}^i \epsilon \mathbb{R}^{M_1 \times M_2}$ of atom i, which serves as input for the fitting network, is defined as:

$$\mathcal{D}^i = \mathcal{G}^{i1}\widetilde{\mathcal{R}}^i(\widetilde{\mathcal{R}}^i)^T\mathcal{G}^{i2} \tag{4.8}$$

which is invariant under translational, rotational, and permutational transformations. We set $\mathcal{G}^{i1} = \mathcal{G}^i$ and take the first $M_2(< M_1)$ columns of $\mathcal{G}^i$ to form $\mathcal{G}^{i2}\epsilon\mathbb{R}^{N_i \times M_2}$. Before using $\mathcal{D}^i$ as input for the fitting network, the matrix is flattened into a vector. The fitting network then predicts the "atomic energy" E_i of atom i.

$$\mathcal{G}^i = \begin{pmatrix} \mathcal{G}^i_{11} & ... & \mathcal{G}^i_{1M_2} & \mathcal{G}^i_{1M_2+1} & ... & \mathcal{G}^i_{1M_1} \\ \mathcal{G}^i_{21} & ... & \mathcal{G}^i_{2M_2} & \mathcal{G}^i_{2M_2+1} & ... & \mathcal{G}^i_{2M_1} \\ ... & ... & ... & ... & ... & ... \\ \mathcal{G}^i_{N_i 1} & ... & \mathcal{G}^i_{N_i M_2} & \mathcal{G}^i_{N_i M_2+1} & ... & \mathcal{G}^i_{N_i M_1} \end{pmatrix}, \tag{4.9}$$

The parameters $\mathbf{w}$ of the embedding and fitting network were trained using the Adam stochastic gradient descent method, optimizing the loss function

$$\mathcal{L}(p_e, p_f, p_v) = \frac{1}{|\mathcal{B}|}\sum_{k\epsilon\mathcal{B}}(p_e|E_k - E_k^{\mathbf{w}}|^2 + p_f|F_k - F_k^{\mathbf{w}}|^2 + p_v||\Xi_k - \Xi_k^{\mathbf{w}}||^2) \tag{4.10}$$

where $\mathcal{B}$ is the minibatch, $|\mathcal{B}|$ is the batch size, k is the index of the training data, and p_e, p_f, p_v are tunable parameters, which were set to zero if the corresponding label was missing from the training data.[14]

4.2. Adding Long Range Interaction

In the approach presented in section 4.1, we now aim to incorporate long-range interactions. To avoid the computational cost of calculating quadratically many pairs, we proceed similarly to FMM described in section 3. Specifically, a grid is constructed, dividing the molecular system into cells that contain the atoms of the molecule. For an atom i, instead of using the pairwise distance to other atoms that are beyond a certain cutoff radius r_c (everything inside r_c is considered local and is already considered in the approach of section 4.1), we employ the center of mass of the grid cells in which these atoms lie. This approach significantly reduces the number of distances that need to be computed while still accounting for the long-range effects.

This technique leverages the idea that, for sufficiently large separations, the precise positions of individual atoms become less important compared to the overall contribution from a region of space. Hence, by utilizing the center of mass of the grid cells, the computational cost scales better compared to the naive all-pairs approach.

Unlike some hierarchical grid-based methods, where cells are recursively subdivided to increase accuracy, we deliberately avoid such recursive refinement in this model. Instead, a fixed grid cell size is used throughout the computation to be more efficient and computationally easier to implement.

4.3. Implementation

The following section will describe the exact procedure and implementation of the idea introduced in section 4.2. First, the preparation of the dataset, including the creation of the grid and the computation of the centers of mass, will be explained in detail. Afterward, the architecture and training process of the employed neural network will be described. The code of the project can be viewed on GitHub: `https://github.com/AaronJacques/long-range-ml-potential`.

4.3.1. Datasets

For the training, the dataset from [16] was used, which was calculated using Density Functional Theory (DFT). The tool used was the "ab initio materials simulations" (aims) from the Fritz Haber Institute, known as "FHI-aims". The dataset provides the coordinates of the atoms, as well as the energy and force labels. The force labels were not used further, as only the energy was used for training. The atomic coordinates are given in Å, and the energy labels in $\frac{\text{kcal}}{\text{mol}}$.

Table 4.1 shows the sizes of the datasets used. The molecules are grouped into small molecules and large molecules. In addition, a maximum of 65,000 training samples were used. For datasets with a larger number of samples, only the first 65,000 samples were utilized. The dataset was randomly split into 80% training, 10% validation, and 10% test sets.

Molecule	Number of Atoms	Available Samples	Used Samples
Paracetamol	20	106,490	65,000
Aspirin	21	211,762	65,000
Ac-Ala3-NHMe	42	85,109	65,000
Stachyose	87	27,272	27,272

Table 4.1.: Number of Training Samples

4.3.2. Preprocessing

During preprocessing, the data is prepared for training. The raw dataset (see section 4.3.1) is further processed to be used directly for training. This step only needs to be performed once. Subsequently, the dataset can be used repeatedly for training.

The following procedure is carried out for each sample of the dataset.

Grid Assignment

The preprocessing process begins with the assignment into different grid cells. Let $R_k = \{\mathbf{r}_1, ..., \mathbf{r}_N\}$ be the positions of the atoms of the k-th molecule and $\mathbf{r}_i = (x_i, y_i, z_i)$ with

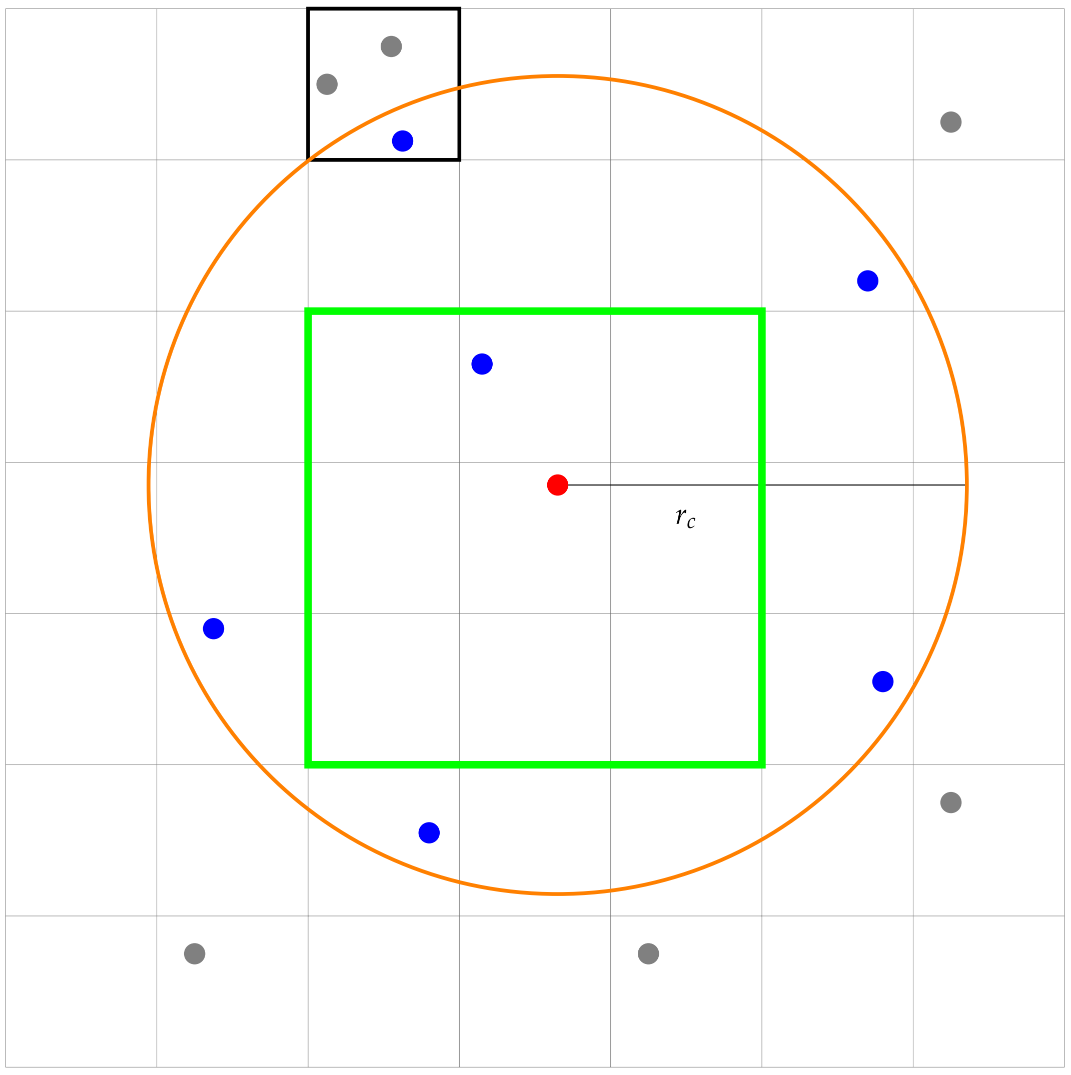


Figure 4.1.: Grid - Illustration of the grid and its parameters in 2D. The currently processed atom is shown in red. The rectangle defined by $l_{\max} = 1$ is depicted in green. The resulting cutoff radius r_c is shown in orange. Atoms within r_c are colored blue and are included in the local matrix. Atoms outside r_c are colored gray and are included in the long-range matrix. The grid cell highlighted in black demonstrates how cells partially covered by r_c are handled. The blue atom is removed from the cell because it lies inside r_c, and the feature vector f^k and the center of mass $\mathbf{r}_m^k$ are formed using only the gray atoms. By removing all blue atoms and subsequently removing all empty cells, the reduced grid for the current atom is obtained.

$i\epsilon\{1, ..., N\}$. The origin of the grid $g_{0,0,0}$ is defined as

$$g_{0,0,0} = (\min_{i\epsilon\{1,...,N\}} x_i, \min_{i\epsilon\{1,...,N\}} y_i, \min_{i\epsilon\{1,...,N\}} z_i) \tag{4.11}$$

which is the minimum along the axes of the atomic positions. The axes of the grid run parallel to the coordinate system, and with the grid length Δg, the grid is uniquely defined. The atoms are then assigned to the grid cells according to their atomic positions. Each grid cell is assigned an x-, a y-, and a z-index, marked by the index tuple (x, y, z), which is $(0, 0, 0)$ for the origin. Let N_g^x be the maximum index in the x-direction, and N_g^y and N_g^z be the corresponding maximum indices in the y- and z-directions, respectively.

Feature Vector & Centre of Mass

Let M be the total number of grid cells calculated above, c_k be the k-th cell with $k\epsilon\{1, ...M\}$ and M_k be the number of atoms in grid cell k. The feature vector $f^k \epsilon \mathbb{N}^{m_{\max}}$ of grid cell k is a vector of counts. Each entry represents how many atoms of a specific element are present in the grid cell. The i-th entry in the vector corresponds to the count of atoms with the atomic number $i+1$. For example, if there is exactly one carbon atom in the cell, $(f^k)_5 = 1$, as carbon has the atomic number 6. All atoms within the molecule that have a higher atomic number than $m_{\max}+1$ are not considered in the feature vector. The centre of mass $\mathbf{r}_m^k$ of the k-th cell is defined as

$$\mathbf{r}_m^k = \frac{\sum_{i=0}^{M_k} z_i^k \cdot \mathbf{r}_i^k}{\sum_{i=0}^{M_k} z_i^k} \tag{4.12}$$

where z_i^k is the atomic number and $\mathbf{r}_i^k$ the position of the i-th atom of grid cell k.

Cutoff Radius

Instead of defining r_c as an input parameter, the maximum number of grid levels $l_{\max}$ is selected, and from this r_c is determined. The 0-th level represents the cell in which the atom itself is located, the 1-st level represents direct neighbors, and so on. Let (j, k, l) be the index tuple of the grid cell in which the i-th atom is located, then

$$\mathcal{G}_i = \{g_{\tilde{j},\tilde{k},\tilde{l}} | \max(|j-\tilde{j}|, |k-\tilde{k}|, |l-\tilde{l}|) \leq l_{\max}\} \tag{4.13}$$

is the set of cells that are included in the local interaction of atom i ($j, \tilde{j}\epsilon\{1, ..., N_g^x\}$ and $k, \tilde{k}\epsilon\{1, ..., N_g^y\}$ and $l, \tilde{l}\epsilon\{1, ..., N_g^z\}$).

The cutoff radius r_c is then determined as follows:

$$r_c = (l_{\max}+1) \cdot \Delta g + \sqrt{2(\frac{\Delta g}{2})^2} = (l_{\max}+1+\frac{1}{\sqrt{2}}) \cdot \Delta g. \tag{4.14}$$

The formula for the cutoff radius is chosen such that all cells included in the local interaction $\mathcal{G}_i$ lie within the cutoff radius. However, there are also cells not included in $\mathcal{G}_i$ that contain atoms within the cutoff radius. These atoms are also included in the local interaction (see next paragraph). Atoms that have already been considered in the local interaction are removed from the cell and are not included in the calculation of $\mathbf{r}_m^k$ and f^k (see Figure 4.1).

Local Matrices

For the local interaction, two matrices are created for each atom i of the molecule using its local environment $\mathcal{N}_{r_c}(i) = \{j | r_{ij} = |\mathbf{r}_{ij}| \leqslant r_c\}$ with N_i being the cardinality of this set. The grid is not relevant for this step. As described in section 4.1, the matrix $\widetilde{\mathcal{R}}^i \epsilon \mathbb{R}^{N_i \times 4}$ is created from the local environment $\mathcal{N}_{r_c}(i)$ of atom i. This matrix is henceforth called $\widetilde{\mathcal{R}}^i_{\text{local}}$ to emphasize its association with the local environment. Let z_i be the atomic number of atom i and z_j with $j \epsilon \mathcal{N}_{r_c}(i)$ be the atomic number of the j-th atom from the local environment. This yields the matrix

$$\mathcal{Z}^i = \begin{pmatrix} z_i & z_0 \\ \dots & \dots \\ z_i & z_j \\ \dots & \dots \\ z_i & z_{N_i} \end{pmatrix}, \tag{4.15}$$

which is sorted such that $(\mathcal{Z}^i)_l$ are the atomic numbers corresponding to the respective distance $(\widetilde{\mathcal{R}}^i)_l$. Thus, the two matrices $\widetilde{\mathcal{R}}^i_{\text{local}}$ and $\mathcal{Z}^i$ are created for each atom i of the molecule. During the calculation of the local matrices, a copy of the grid is created for atom i, and all atoms within the local environment $\mathcal{N}_{r_c}(i)$ are removed from the grid. Subsequently, all empty cells are also removed from the grid. This process leaves only the cells and atoms that are outside the cutoff radius. These cells are used in the next step.

Long Range Matrices

For the long-range interaction of an atom i in the molecule, the idea of the Fast Multipole Method (FMM) was used, where the interaction with each individual long-range partner is not calculated directly but rather with the center of a cell/box. Cells that are not entirely within the cutoff radius r_c are considered part of the long-range interaction. These are precisely the cells/atoms that remain after the calculation of the local matrix (see previous paragraph).

Next, for each atom i of the molecule, the long-range matrix $\mathcal{R}^i_{\text{long}} \epsilon \mathbb{R}^{M_i \times 3}$ is calculated using the reduced grid from the previous section, where $\mathbf{r}_i$ is the position of the i-th atom and $\mathbf{r}_m^k$ the center of mass of the k-th grid cell of the reduced grid, with M_i being the number of grid cells in it.

$$\mathcal{R}^i_{\text{long}} = \{\mathbf{r}_{1i}^T, ..., \mathbf{r}_{ki}^T, ..., \mathbf{r}_{M_i i}^T\}, \mathbf{r}_{ji} = (x_{ji}, y_{ji}, z_{ji}) \tag{4.16}$$

and is constructed by defining the relative coordinates $\mathbf{r}_{ki} \equiv \mathbf{r}_m^k - \mathbf{r}_i$ where k $(1 \leq j \leq M_i)$ is the index of the k-th grid cell. The distance between atom i and the centre of mass of the k-th grid cell is $r_{ki} = \|\mathbf{r}_{ki}\|$.

After constructing the long range matrix $\mathcal{R}^i_{\text{long}}$ it is mapped onto generalized coordinates $\widetilde{\mathcal{R}}^i_{\text{long}} \epsilon \mathbb{R}^{M_i \times 4}$ by transforming each row $\{x_{ji}, y_{ji}, z_{ji}\}$ of $\mathcal{R}^i_{\text{long}}$ into a row of $\widetilde{\mathcal{R}}^i_{\text{long}}$:

$$\{x_{ji}, y_{ji}, z_{ji}\} \mapsto \{s(r_{ji}), \hat{x}_{ji}, \hat{y}_{ji}, \hat{z}_{ji}\}. \tag{4.17}$$

The components are defined as follows $s(r_{ji}) = \frac{1}{r_{ji}}$, $\hat{x}_{ji} = \frac{x_{ji}}{r_{ji}}$, $\hat{y}_{ji} = \frac{y_{ji}}{r_{ji}}$ and $\hat{z}_{ji} = \frac{z_{ji}}{r_{ji}}$.

$$\widetilde{\mathcal{R}}^i_{\text{long}} = \begin{pmatrix} s(r_{1i}) & \hat{x}_{1i} & \hat{y}_{1i} & \hat{z}_{1i} \\ ... & ... & ... & ... \\ s(r_{ji}) & \hat{x}_{ji} & \hat{y}_{ji} & \hat{z}_{ji} \\ ... & ... & ... & ... \\ s(r_{M_i i}) & \hat{x}_{M_i i} & \hat{y}_{M_i i} & \hat{z}_{M_i i} \end{pmatrix}, \tag{4.18}$$

In addition, for each atom i, the feature matrix $\mathcal{F}^i \epsilon \mathbb{R}^{M_i \times (m_{\max}+1)}$ is defined. For the j-th grid cell from the reduced grid of atom i, with the corresponding j-th row $(\widetilde{\mathcal{R}}^i_{\text{long}})j = (s(rji), \hat{x}_{ji}, \hat{y}_{ji}, \hat{z}_{ji})$ from the long-range matrix and the feature vector f^j, the j-th row of the feature matrix is:

$$(\mathcal{F}^i)_j = (z_i, (f^j)_1, ..., (f^j)_{m_{\max}}), \tag{4.19}$$

where z_i is the atomic number of the i-th atom. Thus, the complete feature matrix $\mathcal{F}^i$ is formed as follows:

$$\mathcal{F}^i = \begin{pmatrix} z_i & (f^1)_1 & ... & (f^1)_{m_{\max}} \\ ... & ... & ... & ... \\ z_i & (f^j)_1 & ... & (f^j)_{m_{\max}} \\ ... & ... & ... & ... \\ z_i & (f^{M_i})_1 & ... & (f^{N_i})_{m_{\max}} \end{pmatrix}, \tag{4.20}$$

Final Model Input

The network receives a total of four matrices as final input: for the local environment, $\widetilde{\mathcal{R}}^i_{\text{local}}$ and $\mathcal{Z}^i$, and for the long-range interaction, $\widetilde{\mathcal{R}}^i_{\text{long}}$ and $\mathcal{F}^i$. For technical reasons, these matrices are padded with zeros so that all $\widetilde{\mathcal{R}}^i_{\text{local}}$ and $\mathcal{Z}^i$ have the same first dimension $N^{\text{local}}_{\max}$. Thus, $\widetilde{\mathcal{R}}^i \epsilon \mathbb{R}^{N^{\text{local}}_{\max} \times 4}$ and $\mathcal{Z}^i \epsilon \mathbb{N}^{N^{\text{local}}_{\max} \times 2}$, with $N_i \leq N^{\text{local}}_{\max}$ for all $i\epsilon\{1, ..., N\}$. Similarly, both long-range matrices are padded with $N^{\text{long}}_{\max}$: $\widetilde{\mathcal{R}}^i \epsilon \mathbb{R}^{N^{\text{long}}_{\max} \times 4}$ and $\mathcal{F}^i \epsilon \mathbb{N}^{N^{\text{long}}_{\max} \times (m_{\max}+1)}$, with $M_i \leq N^{\text{long}}_{\max}$ for all $i\epsilon\{1, ..., N\}$.

4.3.3. Architecture

The network is divided into three parts: an embedding network for the local matrices, an embedding network for the long-range matrices (see Figure 4.3), and the energy network (see Figure 4.4).

Both embedding networks and the energy network are based on the Dense-Res-Block (see Figure 4.2), inspired by [7]. The skip connection in this block enhances gradient flow during training [7]. Each block receives a parameter N, which determines the output size of the first two dense layers. The third dense layer adjusts the vector length to D, matching the input length to the block to enable element-wise addition. To prevent overfitting and stabilize training, a dropout layer and two layer normalization layers are included, as dense layers are used instead of convolutional layers.

The embedding network uses only the first column of the input matrix $\widetilde{\mathcal{R}}^i$, denoted as $s_i = (s(r_{1i}), \ldots, s(r_{N_{\text{max}}i})) \epsilon \mathbb{R}^{N_{\text{max}}}$. Unlike the original approach, a single large embedding network is used for all atom species combinations α_1 and α_2, instead of separate networks. This network also incorporates atomic species information through the matrices $\mathcal{Z}^i$ (local) and $\mathcal{F}^i$ (long range). After applying several Dense-Res-Blocks, the output is transformed into the embedding matrix $\mathcal{G}^i \epsilon \mathbb{R}^{N_{\text{max}} \times K_1}$, where K_1 is an adjustable parameter.

We set $\mathcal{G}^{i1} = \mathcal{G}^i$ and obtain $\mathcal{G}^{i2}$ by selecting the first K_2 ($< K_1$) columns of $\mathcal{G}^i$ (see Equation 4.9). From this, the feature matrix $\mathcal{D}^i \epsilon \mathbb{R}^{K_1 \times K_2}$ is constructed, as shown in Equation 4.8.

The energy network receives the feature matrices $\mathcal{D}^i$local and $\mathcal{D}^i$long to generate the 'atomic energy' E_i of the i-th atom in the molecule, which in itself has no physical meaning. To obtain the total energy of the molecule, the entire network must be applied to all atoms i and the resulting energies summed.

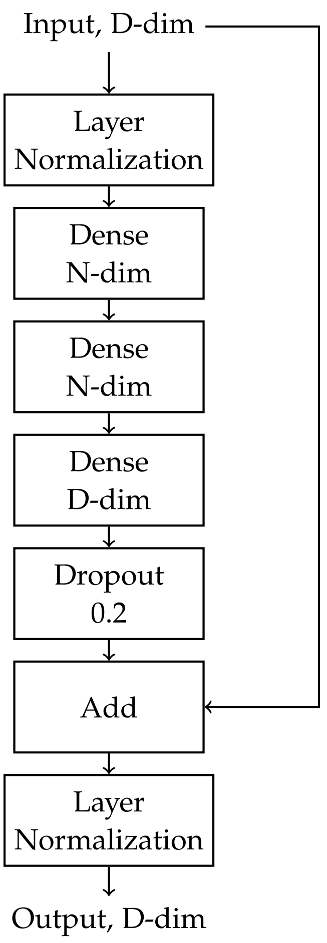


Figure 4.2.: Dense-Res-Block with N filters - After each Dense block, an ELU activation function is applied. N is a parameter given to the block, while D depends on the input shape of the block. The third Dense layer assures that both inputs of the add function have the same length. The output dimension and the dropout probability are written below the layer.

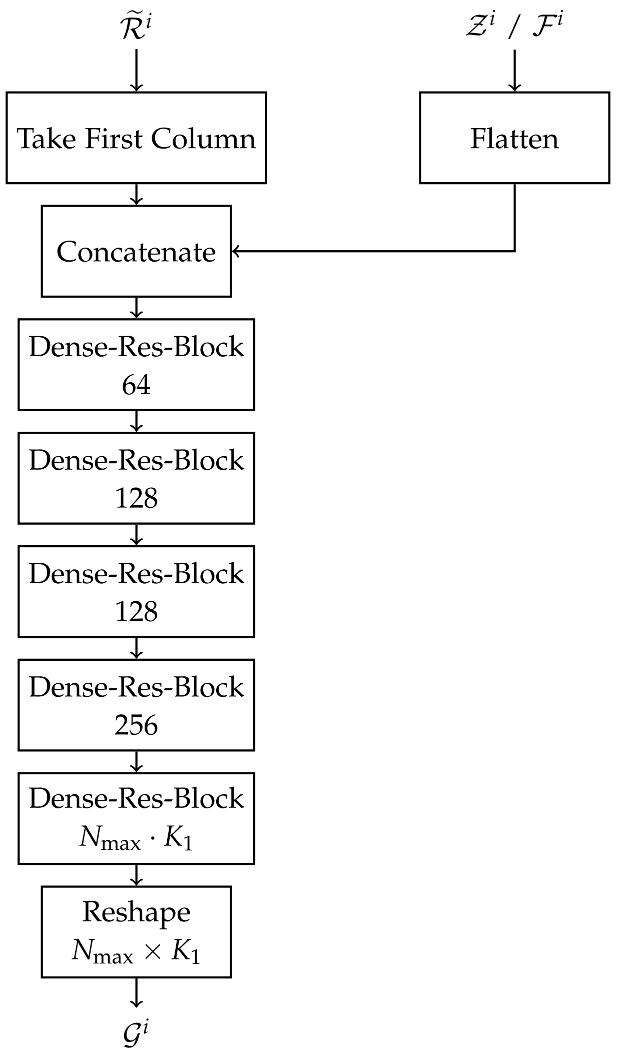

$\widetilde{\mathcal{R}}^i$
$\mathcal{Z}^i$ / $\mathcal{F}^i$
Take First Column
Flatten
Concatenate
Dense-Res-Block 64
Dense-Res-Block 128
Dense-Res-Block 128
Dense-Res-Block 256
Dense-Res-Block $N_{\max} \cdot K_1$
Reshape $N_{\max} \times K_1$
$\mathcal{G}^i$

Figure 4.3.: Embedding Network - Either the local matrices $\widetilde{\mathcal{R}}^i_{\text{local}}$ and $\mathcal{Z}^i$ or the long range matrices $\widetilde{\mathcal{R}}^i_{\text{long}}$ and $\mathcal{F}^i$ are used as input to the embedding network. First, the first column of the matrix $\widetilde{\mathcal{R}}^i$ is taken, corresponding to the vector $s_i = (s(r_{1i}), \ldots, s(r_{N_{\max}i})) \epsilon \mathbb{R}^{N_{\max}}$. Both $\mathcal{Z}^i$ and $\mathcal{F}^i$ are first transformed into vectors by concatenating the rows of the matrix and then concatenated with vector s_i. After that, five Dense-Res-Blocks are applied in sequence, with the last block used to produce the output dimension. Finally, the output is reshaped into the matrix $\mathcal{G}^i \epsilon \mathbb{R}^{N_{\max} \times K_1}$. K_1 is a tunable parameter.

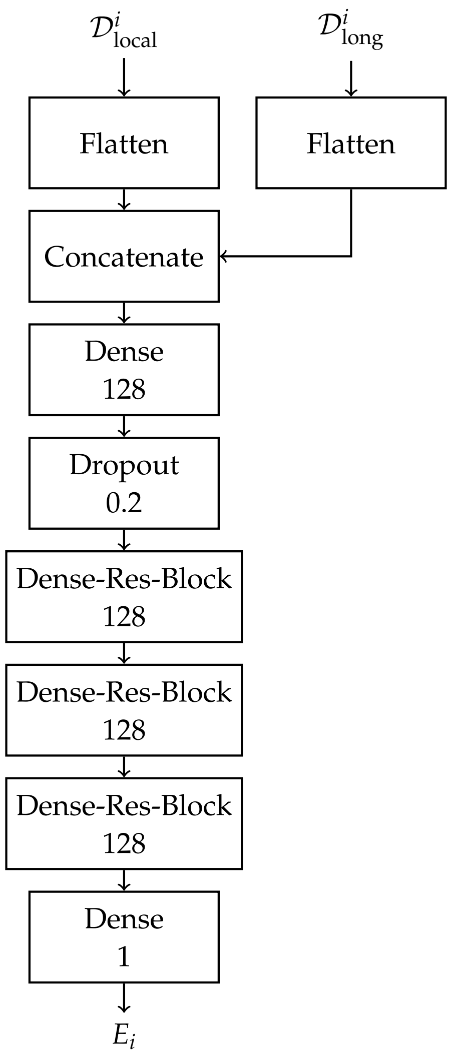


Figure 4.4.: Energy Network - The energy network receives the feature matrices $\mathcal{D}^i_{\text{local}}$ and $\mathcal{D}^i_{\text{long}}$ as input, which is invariant to translation, rotation, and permutation. These matrices are first flattened into vectors and then concatenated. The first dense layer produces an output of length 128 and uses the ELU activation function. A dropout layer with a dropout probability of 0.2 is placed before the dense blocks. The final layer uses a linear activation function to produce the output.

$$\begin{tikzcd}
\mathcal{Z}^i \arrow[d] & \widetilde{\mathcal{R}}^i_{\text{local}} \arrow[dl] \arrow[ddd] & & \widetilde{\mathcal{R}}^i_{\text{long}} \arrow[dr] \arrow[ddd] & \mathcal{F}^i \arrow[d] \\
\text{Local Embedding} \arrow[d] & & & & \text{Long Embedding} \arrow[d] \\
\mathcal{G}^i_{\text{local}} \arrow[dr] & & & & \mathcal{G}^i_{\text{long}} \arrow[dl] \\
& \mathcal{D}^i_{\text{local}} = \mathcal{G}^{i1}_{\text{local}} \widetilde{\mathcal{R}}^i_{\text{local}} (\widetilde{\mathcal{R}}^i_{\text{local}})^T \mathcal{G}^{i2}_{\text{local}} \arrow[dr] & & \mathcal{D}^i_{\text{long}} = \mathcal{G}^{i1}_{\text{long}} \widetilde{\mathcal{R}}^i_{\text{long}} (\widetilde{\mathcal{R}}^i_{\text{long}})^T \mathcal{G}^{i2}_{\text{long}} \arrow[dl] & \\
& & \text{Energy Network} \arrow[d] & & \\
& & E_i & &
\end{tikzcd}$$

Figure 4.5.: The architecture of the full network

$$\begin{tikzcd}
\mathcal{Z}^i \arrow[d] & & \widetilde{\mathcal{R}}^i_{\text{local}} \arrow[dll] \arrow[dddl] \\
\text{Local Embedding} \arrow[d] & & \\
\mathcal{G}^i_{\text{local}} \arrow[dr] & & \\
& \mathcal{D}^i_{\text{local}} = \mathcal{G}^{i1}_{\text{local}} \widetilde{\mathcal{R}}^i_{\text{local}} (\widetilde{\mathcal{R}}^i_{\text{local}})^T \mathcal{G}^{i2}_{\text{local}} \arrow[d] & \\
& \text{Energy Network} \arrow[d] & \\
& E_i &
\end{tikzcd}$$

Figure 4.6.: The architecture of the network without long-range interactions - Obtained by removing the right part of figure 4.5.

4.3.4. Training

For training the parameters **w** of both the embedding networks and the energy network, the Stochastic Gradient Descent method with the Adam optimizer was employed. The following loss function was minimized:

$$\mathcal{L} = \frac{1}{|\mathcal{B}|} \sum_{k \epsilon \mathcal{B}} |E_k - E_k^{\mathbf{w}}|^2 \tag{4.21}$$

where $\mathcal{B}$ represents the minibatch, $|\mathcal{B}|$ is the batch size, and k indexes the training data. Here, E_k denotes the true total energy of the molecule, and $E_k^{\mathbf{w}}$ is the predicted total energy generated by the network. The predicted total energy is obtained by repeatedly applying the network and summing the resulting 'atomic energies'.

During training, the learning rate was exponentially decayed according to the formula:

$$\eta(t) = \eta_{\text{init}} \cdot 0.97^{\frac{t}{s}} \tag{4.22}$$

where t is the current training step, η_{init} is the initial learning rate, and s is the decay step.

For the training, $|\mathcal{B}| = 1$ was used.

5. Results: Benchmark Comparison to Conventional Neural Networks

In this chapter, we explore the effectiveness of the approach in predicting the energy of various molecules. Specifically, we compare the performance of the network when incorporating long-range interactions (see Figure 4.5) against a version without these interactions (see Figure 4.6).

For training, the datasets described in Section 4.3.1 were used.

The parameters used in the preprocessing when creating the grid are listed in table 5.1, and the parameters used for training are listed in table 5.2.

Molecule	Δg (Å)	$l_{\max}$	r_c (Å)	r_{cs} (Å)
Paracetamol	1	6	7.71	7.51
Aspirin	1	6	7.71	7.51
Ac-Ala3-NHMe	2	2	7.41	7.21
Stachyose	2	4	11.41	11.21

Table 5.1.: Parameters used in preprocessing when creating the grid

Molecule	$N_{\max}^{\text{local}}$	$N_{\max}^{\text{long}}$	K_1^{local}	K_1^{long}	K_2^{local}	K_2^{long}	η_{init}	s	epochs
Paracetamol	19	6	19	6	4	1	5×10^{-5}	2.08×10^5	200
Aspirin	20	4	20	4	4	1	1×10^{-4}	2.08×10^5	200
Ac-Ala3-NHMe	41	20	41	20	4	2	5×10^{-5}	2.08×10^5	150
Stachyose	86	28	30	6	4	2	5×10^{-5}	8.727×10^4	130

Table 5.2.: Parameters used for training

$N_{\max}^{\text{local}} = K_1^{\text{local}}$ and $N_{\max}^{\text{long}} = K_1^{\text{long}}$ were always chosen. K_2 was set significantly smaller than K_1. s was chosen to be four times the size of the training dataset. The values were determined through hyperparameter tuning.

All models were trained for the specified epochs, and the epoch with the best Mean Square Error (MSE) on the validation dataset was selected for benchmarking on the test dataset.

The Mean Absolute Error (MAE) in $\frac{\text{kcal}}{\text{mol}}$ on the test dataset are presented in table 5.3.

Molecule	With long-range	Only local	Relative Improvement (%)
Paracetamol	2.465	3.024	22.68
Aspirin	1.670	1.657	-0.78
Ac-Ala3-NHMe	4.039	4.087	1.17
Stachyose	5.841	5.861	0.34

Table 5.3.: Comparison of MAE loss - MAE loss in $\frac{\text{kcal}}{\text{mol}}$. Relative improvement is shown in %.

As the data shows, adding long-range interactions does not significantly improve the model predictions. The only noticeable improvement is observed with Paracetamol. This could be due to the lack of systematic hyperparameter tuning, leading to the possibility that only Paracetamol benefited from randomly finding very good parameters. The outcome is particularly sensitive to the choice of parameters K_1 and K_2.

Figure 5.3 shows the distribution of weights in the network. It can be observed that for all molecules, the distribution is almost identical with and without long-range interaction. However, in the network with long-range interaction, more weights are 0 or close to 0, and fewer weights with a larger magnitude. This suggests that the network is learning to ignore the long-range contributions and focus only on the local contributions, which results in the MAE loss being nearly the same for both networks.

Figures 5.1 and 5.2 illustrate the MSE energy loss during training for Paracetamol. Since the loss is very large at the beginning of the training, the first 19 epochs have been removed from the graph.

The validation loss fluctuates significantly, especially at the beginning, which is most likely due to the small batch size. Additionally, it can be observed that the network without long-range interaction converges much faster, as it has fewer parameters to learn.

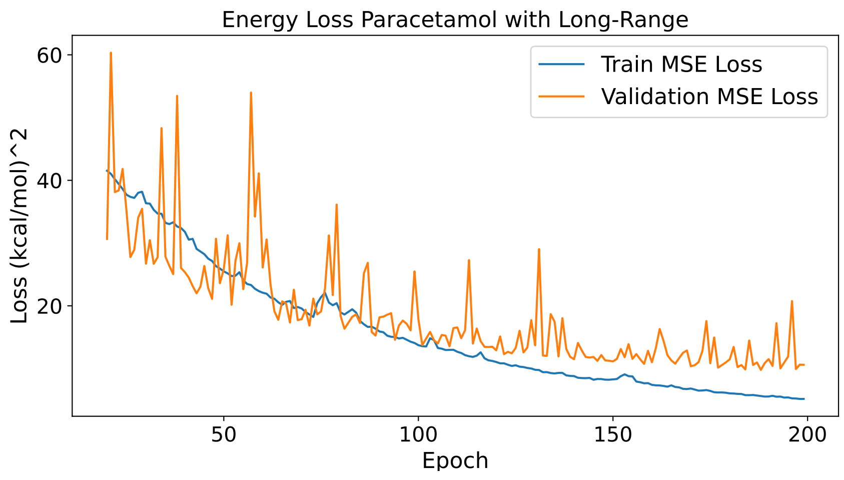


Figure 5.1.: Energy Loss with Long Range Interactions - MSE Energy Loss of Paracetamol with Long Range Interaction

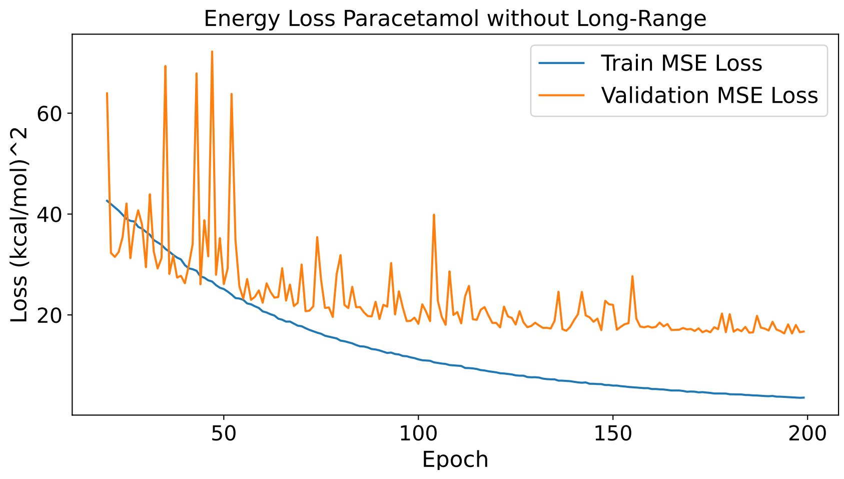
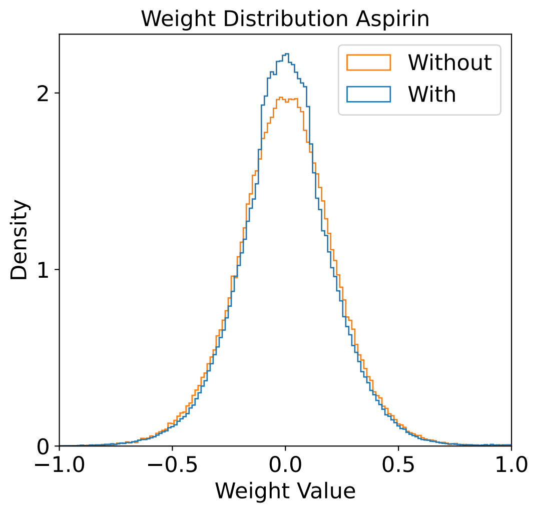
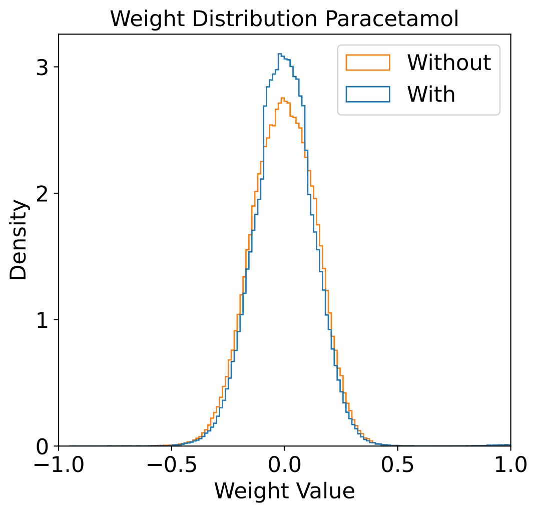


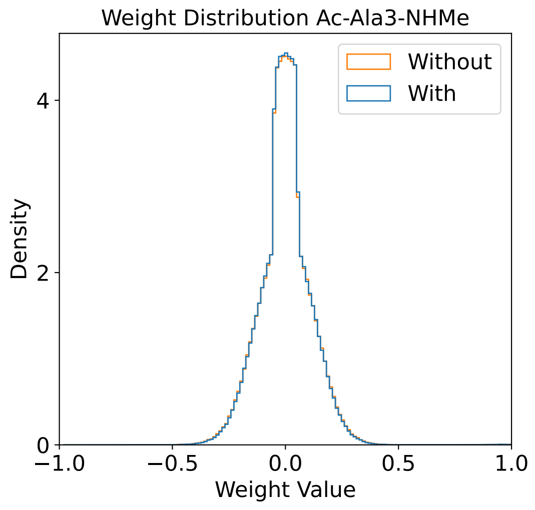
Figure 5.2.: Energy Loss without Long Range Interactions - MSE Energy Loss of Paracetamol without Long Range Interaction



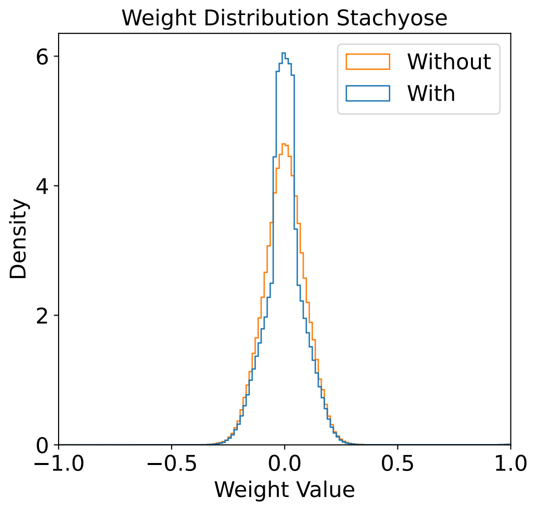
(a) Aspirin



(b) Paracetamol



(c) Ac-Ala3-NHMe



(d) Stachyose

Figure 5.3.: Weight distributions of the model. Blue: With long-range interactions. Orange: Without long-range interactions. Weights smaller than -1 or greater than one have been disregarded. 150 bins have been used to plot the distribution.

6. Discussion

Table 6.1 presents a comparison of the Mean Absolute Error (MAE) loss values obtained from different computational approaches for predicting molecular properties. It is evident that sGDML [17] consistently provides significantly better predictions for all the molecules studied. Notably, only for Stachyose does the MAE of this approach come close to that of sGDML. This observation suggests that this method, which incorporates long-range interactions during preprocessing, may perform better with larger molecules. The additional computational effort required for modeling long-range interactions might thus only be justified for complex molecular systems where long-range interactions play a more significant role. This warrants further investigation to fully understand the scalability and potential benefits of our approach for larger molecules. While the architecture is capable of handling larger molecules, there were no sufficiently large datasets available for testing at this time.

Molecule	This approach with long-range	DeepPot-SE [14]	sGDML [17]
Paracetamol	2.465	-	0.113
Aspirin	1.670	0.155	0.047
Ac-Ala3-NHMe	4.039	-	0.391
Stachyose	5.841	-	4.002

Table 6.1.: MAE loss (kcal/mol) comparison of different methods - All approaches used the dataset from [16]. sGDML consistently performs the best across all tested molecules.

DeepPot-SE [14], another method, provides competitive results for Aspirin but lacks data for the other molecules in this work. Its MAE for Aspirin is lower than that of our approach but higher than sGDML, indicating that while DeepPot-SE is effective, sGDML still holds an advantage in accuracy. The likely reason for the inferior performance compared to the original method is the use of a single large embedding network for all atom types rather than specific combinations for different atom types and the lack of systematic hyperparameter tuning.

In conclusion, while sGDML currently outperforms our approach, especially for smaller molecules, our method shows potential for larger molecules due to its incorporation of long-range interactions. Continued research and development could enhance its performance, making it a valuable tool in computational chemistry and materials science.

7. Outlook: What can be improved?

In this chapter, we reflect on the limitations and potential areas for improvement in the current work. While significant progress has been made, there are still several aspects that could be optimized or further explored to enhance the accuracy, efficiency, and applicability of the methods developed.

7.1. Smooth Descriptor

The overall approach is divided into two parts: the local interaction and the long-range interaction. The descriptor for the local interaction is smooth; however, the descriptor for the long-range interaction is not. In the long-range descriptor, there are two instances where the matrix $\widetilde{\mathcal{R}}^i_{\text{long}}$ undergoes abrupt changes due to slight changes in an atom's position: the transition between local and long-range interactions and the transition between two grid cells. Improving the method at these points could potentially lead to further enhancements in prediction accuracy.

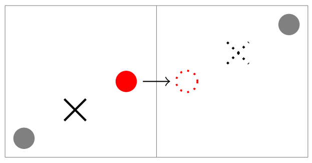


Figure 7.1.: Non-Smooth Switching of Grid Cells

When the position of the red atom (Figure 7.1) is slightly altered, causing it to move into a different grid cell, the centers of mass of both cells change abruptly. Before the shift, the center of mass of the left cell is at the marked cross, while the center of mass of the right cell is at the only atom (gray). After the shift, the center of mass in the left cell jumps to the gray atom, and in the right cell, it jumps to the dotted cross. A similar situation occurs when an atom moves out of the cut-off radius and is subsequently included in the long-range interaction. This transition also leads to abrupt changes in the descriptors, affecting the continuity of the interaction modeling.

7.2. Hierarchical Grid

Instead of using a fixed grid size Δg, the matrices for the long-range interactions could be generated using the hierarchical approach, as in the FMM (section 3). This would have the advantage that grid cells further away from an atom would be larger than those nearby. In this way, the spatial structure would be better incorporated into the descriptor, and the total number of grid cells required for the computation would be reduced.

7.3. Larger Datasets

An additional improvement could involve generating larger datasets through DFT calculations. Incorporating new DFT-generated data would allow for the exploration of a broader range of molecular systems and enhance the representation of the selected molecules, ultimately providing the model with more diverse and comprehensive training data.

7.4. Larger Batch Size

Due to technical limitations, a batch size of only one was used for the entire training process, which led to significant fluctuations in the validation loss during training. This made the training numerically unstable. Using a larger batch size could help prevent this instability.

7.5. Systematic Hyperparameter Tuning

Due to limited computational resources, systematic hyperparameter tuning could not be performed. It is possible that the loss of the models with long-range interaction could be further reduced. In particular, it would be valuable to examine how the loss behaves for different parameters Δg and $l_{\max}$.

A. Appendix

This chapter presents the additional loss graphs.

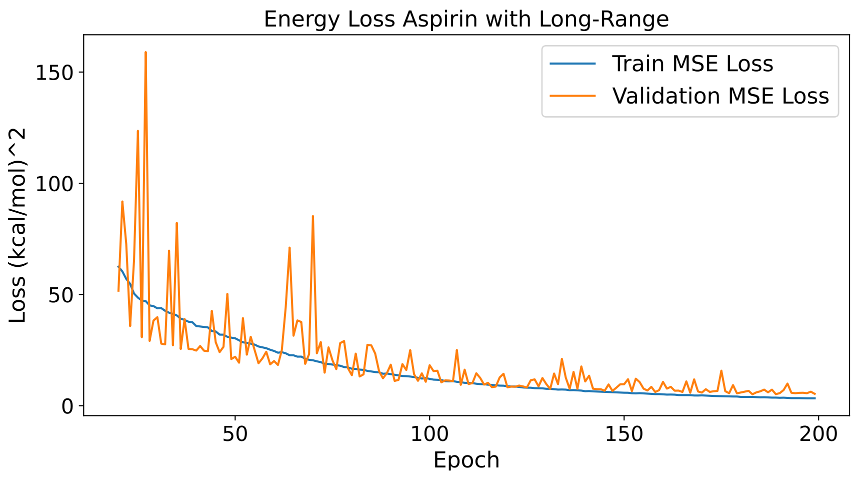


Figure A.1.: MSE Energy Loss with Long-Range Interactions Aspirin

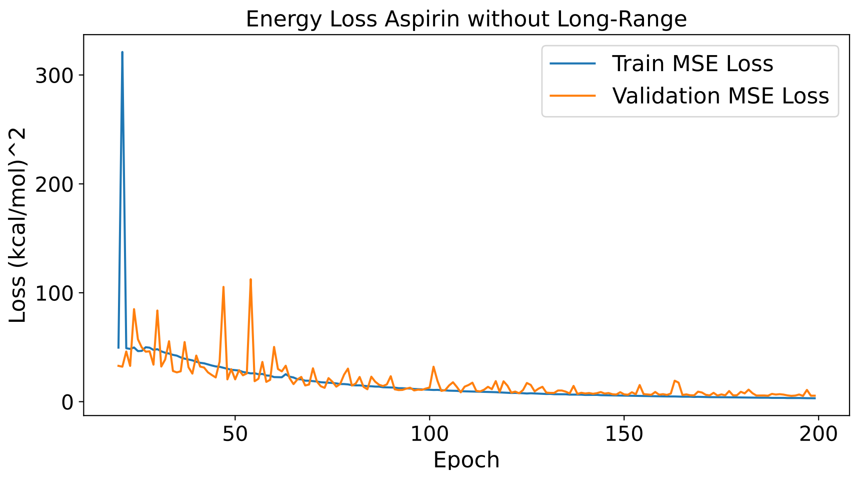


Figure A.2.: MSE Energy Loss without Long-Range Interactions Aspirin

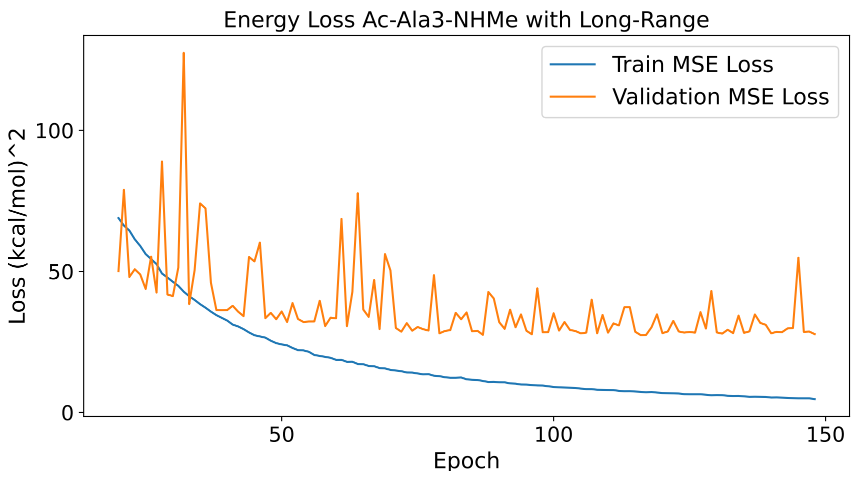


Figure A.3.: MSE Energy Loss with Long-Range Interactions Ac-Ala3-NHMe

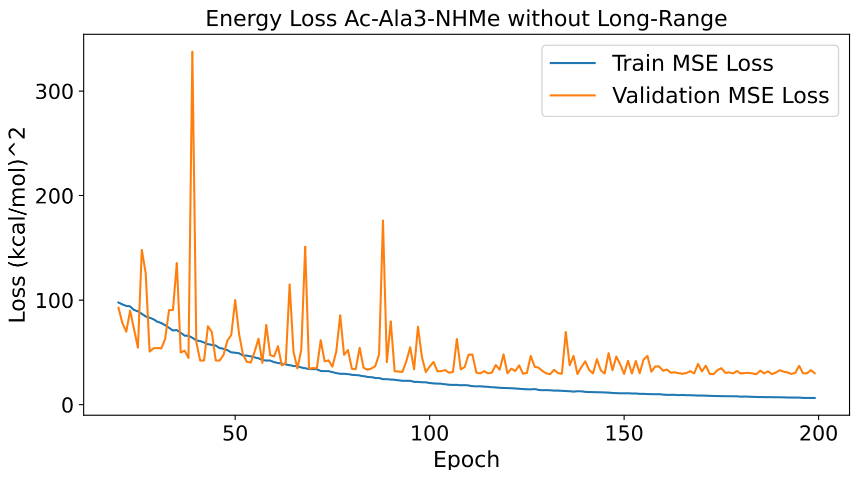


Figure A.4.: MSE Energy Loss without Long-Range Interactions Ac-Ala3-NHMe

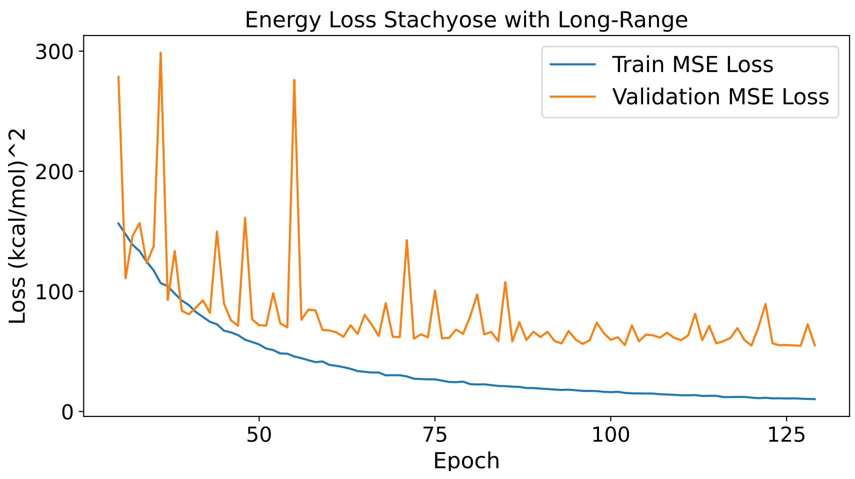


Figure A.5.: MSE Energy Loss with Long-Range Interactions Stachyose

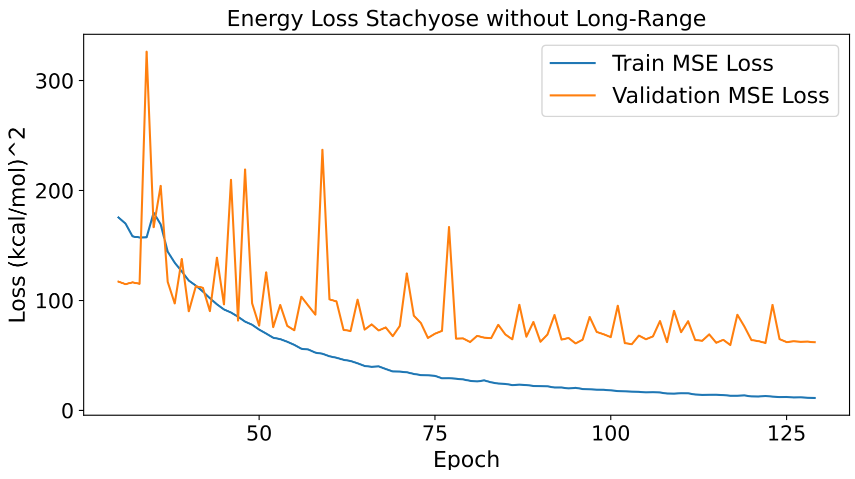


Figure A.6.: MSE Energy Loss without Long-Range Interactions Aspirin

List of Figures

List of Tables

Bibliography

[1] Stanford. *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io (visited on 07/02/2024).

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[3] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106.

[4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[5] J. Lei Ba, J. R. Kiros, and G. E. Hinton. "Layer normalization". In: *ArXiv e-prints* (2016), arXiv–1607. URL: https://arxiv.org/pdf/1607.06450.

[6] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *Under Review of ICLR2016 (1997)* (Nov. 2015).

[7] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: (June 2016). URL: https://doi.org/10.48550/arXiv.1512.03385.

[8] K. Vu, J. C. Snyder, L. Li, M. Rupp, B. F. Chen, T. Khelif, K.-R. Müller, and K. Burke. "Understanding kernel ridge regression: Common behaviors from simple functions to density functionals". In: *International Journal of Quantum Chemistry* 115.16 (2015), pp. 1115–1128. DOI: https://doi.org/10.1002/qua.24939. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.24939.

[9] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32 (2019), pp. 4–24. URL: https://api.semanticscholar.org/CorpusID:57375753.

[10] F. Noé, A. Tkatchenko, K.-R. Müller, and C. Clementi. "Machine Learning for Molecular Simulation". In: *Annual Review Physical Chemistry* 71 (2020), pp. 90–361.

[11] T. Wen, L. Zhang, H. Wang, W. E, and D. J. Srolovitz. "Deep potentials for materials science". In: *Materials Futures* 1.2 (May 2022), p. 022601. DOI: 10.1088/2752-5724/ac681d.

[12] R. Drautz. “Atomic cluster expansion for accurate and transferable interatomic potentials”. In: *Phys. Rev. B* 99 (1 Jan. 2019), p. 014104. DOI: 10.1103/PhysRevB.99.014104. URL: https://link.aps.org/doi/10.1103/PhysRevB.99.014104.

[13] H. Wang, L. Zhang, J. Han, and W. E. “DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics”. In: *Computer Physics Communications* 228 (2018), pp. 178–184. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2018.03.016. URL: https://www.sciencedirect.com/science/article/pii/S0010465518300882.

[14] L. Zhang, J. Han, H. Wang, W. Saidi, R. Car, and W. E. “End-to-end Symmetry Preserving Inter-atomic Potential Energy Model for Finite and Extended Systems”. In: 31 (2018). Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/e2ad76f2326fbc6b56a45a56c59fafdb-Paper.pdf.

[15] L. Ying. “A pedestrian introduction to fast multipole methods”. In: *Science China Mathematics* 55.5 (May 2012), pp. 1043–1051. URL: https://doi.org/10.1007/s11425-012-4392-0.

[16] *sGDML Symmetric Gradient Domain Machine Learning*. URL: http://www.sgdml.org (visited on 07/28/2024).

[17] S. Chmiela, V. Vassilev-Galindo, O. T. Unke, A. Kabylda, H. E. Sauceda, A. Tkatchenko, and K.-R. Müller. “Accurate global machine learning force fields for molecules with hundreds of atoms”. In: *Science Advances* 9.2 (2023), eadf0873. DOI: 10.1126/sciadv.adf0873.